

TIBCO® Object Service Broker for z/OS

External Environments

*Software Release 6.0
July 2012*

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

Copyright © 1999-2012 TIBCO Software Inc. ALL RIGHTS RESERVED.

TIBCO Software Inc. Confidential Information

Contents

Preface	xvii
Related Documentation	xviii
TIBCO Object Service Broker Documentation	xviii
Typographical Conventions	xxiii
Connecting with TIBCO Resources	xxv
How to Join TIBCOCommunity	xxv
How to Access All TIBCO Documentation	xxv
How to Contact TIBCO Support	xxv
Chapter 1 Introduction	1
TIBCO Object Service Broker Architecture	2
Client Services Layer	2
Execution Environment	3
Data Object Broker	3
Accessing TIBCO Object Service Broker from an External Environment	5
What is an External Environment?	5
Facilities Available for Interfacing with External Environments	6
Stages to Setting Up and Processing Within TIBCO Object Service Broker	8
Main Stages	8
Interaction with TIBCO Object Service Broker and its External Environments	8
Chapter 2 The TIBCO Object Service Broker Client Model	11
Overview	12
TIBCO Object Service Broker Clients	12
User Clients	13
TIBCO Object Service Broker SDK (C/C++) Client	14
TIBCO Object Service Broker SDK (Java) Client	14
Same Environments and Address Spaces	14
TIBCO Object Service Broker Client Styles	15
What Determines the Client Style?	15
Seamless or Non-Seamless Client Styles	15
External User or External Transaction Security	16
Display or Non-Display Clients	16
Conversational or Non-Conversational Clients	17
Client Style Summary	17
Setting Up the User Profile for Seamless Clients	18

Setting up A User Profile	18
Guidelines for Development Environments	18
Chapter 3 Setting Execution Environment Parameters	19
Usage of the Execution Environment Parameters	20
Purpose	20
Precedence of Values	20
Determining Session Characteristics	21
Where to Specify Parameters for Single-Session Clients	21
Where to Specify Parameters for Multiple-Session Clients	22
Available Execution Environment Parameters	25
Parameters Specific to the Execution Environment	25
Parameters for Your Session	26
Specifying Session Parameters Using an Input File or a CLIST	27
Format of the Input File	27
Example of Instream Parameter List Using HRNIN in JCL	27
Reducing Session Resources	28
Bypassing the Workbench	28
Operational Characteristics	28
Changing the Invocation Options	29
Non-seamless CICS Client	29
Chapter 4 TIBCO Object Service Broker Sessions Under z/OS Batch	31
How to Run Batch Applications	32
Invocation	32
Using the TIBCO Object Service Broker Supplied Batch Client Program	32
Using A Customized (User) Batch Client	34
How to Set Session Parameters	36
Establishing Session Parameter Values	36
Available DDnames	37
Using an Instream Parameter List	38
How to Manipulate Data in a TIBCO Object Service Broker Batch Client Session	39
Passing Data to TIBCO Object Service Broker Batch Sessions	39
Returning from the Batch Client	39
Returning Data to a User Batch Client	40
Using External Routines	40
Chapter 5 TIBCO Object Service Broker Sessions Under TSO	41
How to Run TSO Applications	42
Invocation	42
Using the TIBCO Object Service Broker Supplied TSO Client Program	42
Using a Customized (User) TSO Client	44

How to Set Session Parameters	46
Establishing Session Parameter Values	46
Using a CLIST to Invoke TIBCO Object Service Broker	47
Specifying or Concatenating a Load Library	48
USER CLIST Distributed with TIBCO Object Service Broker	48
How to Manipulate Data in TSO Client Sessions	49
Passing Data to TIBCO Object Service Broker TSO Sessions	49
Returning from the TSO Client	49
Using External Routines	50
 Chapter 6 TIBCO Object Service Broker Sessions Under the Native Execution Environment 51	
Overview of the Native Execution Environment	52
What is the Native Execution Environment?	52
VTAM and TIBCO Object Service Broker Interaction	52
How to Set Session Parameters	54
Where to Specify Session Parameters	54
Available DDnames	55
Print Destination Restrictions	55
Establishing a TIBCO Object Service Broker VTAM LU2 Session	55
Manipulating Data in VTAM LU2 Client Sessions	57
Passing and Returning Data	57
Determining the Next Step	57
Calling External Routines	57
 Chapter 7 Using the TIBCO Service Gateway for CICS59	
How to Run CICS Applications	60
CICS Client Programs	60
Using the TIBCO Object Service Broker Supplied CICS Modules	60
Using a Customized (User) CICS Client	62
Session Initiation and Termination	64
What Starts and Terminates an Execution Environment?	64
Methods of Session Initiation and Termination	64
Replacing a CICS Transaction with TIBCO Object Service Broker Rules	65
Selecting a TIBCO Object Service Broker CICS Client Program	66
TIBCO Object Service Broker CICS Client Programs	66
Choosing the Right TIBCO Object Service Broker CICS Client Program	67
How to Set Session Parameters	68
Where to Specify CICS Execution Environment Parameters	68
Available DDnames	69
Print Destination Restrictions	69
Synchronization of VSAM Files	69
Starting TIBCO Object Service Broker Sessions	70

Using the Command Line to Start a Session	70
Using EXEC CICS START to Start a Session	70
Using EXEC CICS START to Start a Session with Channel	73
Using EXEC CICS LINK to Start a Session	77
Using EXEC CICS LINK to Start a Session with Channel	79
Using EXEC CICS XCTL to Start a Session	81
Using EXEC CICS XCTL to Start a Session with Channel	82
Passing the COMMAREA Between a TIBCO Object Service Broker CICS Client and a Session	85
Non-Seamless COMMAREA	85
Seamless COMMAREA	86
Retrieving the COMMAREA in a Rule	86
Error Messages in the COMMAREA	87
How Can Data Be Returned	88
Returning Data From TIBCO Object Service Broker to CICS	88
Steps to Returning an Occurrence	88
Using MAP Tables to Return Data	88
Using \$SETENVCOMMAREA to Return Data	89
Performing CICS Functions at Session End	90
Starting a CICS Transaction	90
Transferring to a CICS Program	90
Calling External Routines	91
Calling an External CICS Routine	91
Requirements for Calling an External CICS Routine	91
Restrictions for Calling an External CICS Routine	91
Calling An External Routine With OS Linkage	92
CICS Channels and Containers in the TIBCO Object Service Broker CICS Session Environment	93
CICS Channel and Container Tools	93
Channel Scope	95
Predefined Container Names	95
Chapter 8 Using the TIBCO Service Gateway for IMS TM	97
How to Run IMS TM Applications	98
Functional Overview	98
Using TIBCO Object Service Broker IMS TM Client Programs	98
How a Client Session is Established	98
IMS TM and TIBCO Object Service Broker Interaction	100
Replacing IMS TM Programs with TIBCO Object Service Broker Rules	100
Selecting a TIBCO Object Service Broker IMS TM Program Style	101
TIBCO Object Service Broker IMS TM Client Programs	101
Choosing the Right TIBCO Object Service Broker IMS TM Client Program	102
Starting a TIBCO Object Service Broker Session	103
How to Set Session Parameters	103

Using the IMS TM Terminal to Start a Session	103
Usage of the Supplied Trancode	103
Using Message Formatting Services (MFS) to Start a Session	104
Program-to-Program Message Switching to Start a Session	104
Terminal Changes at Session Startup	105
Extended Terminal Support	105
Additional Terminal Capabilities	105
PF Key Changes	105
Passing Data to TIBCO Object Service Broker IMS TM Sessions	106
Using MAP Tables to Access Data	106
Using the \$GETENVCOMMAREA Tool to Access Data	106
Input Message Segment Overview	108
Message Segment Types	108
S6BIMxC1 Client Program Input Message Format	109
S6BIMxC2 Client Program Input Message Format	109
S6BIMxN1 Client Program Input Message Format	110
S6BIMxN2 Client Program Input Message Format	111
Returning Data from TIBCO Object Service Broker to IMS TM	112
Example using \$SETENVCOMMAREA	112
Passing Control to an IMS Transaction at Session End	114
What are the Allowable Options for Passing Control	114
What to Use to Direct the Destination of Message Segments	114
Non-Conversational MFS Output	114
Non-Conversational Program-to-Program Switch	115
Conversational Deferred Message Switch	115
Conversational Immediate Message Switch	116
Ensuring Message Queue/Database Consistency	117
TIBCO Object Service Broker Supplied Facility	117
@IMSDCTRXS Table	117
Sample Rules for Processing	117
Customizing TIBCO Object Service Broker IMS TM Client Programs	119
Using a Session Exit Routine	119
Where to Enter the Exit Routine	119
Exit Routine Indicators	119
Getting Access to IMS TM Data	120
Overview of the IMS TM Logger Exit	120
Format 1 DATAIN	120
Logger Exit Processing	122
Example of Messages	123
Chapter 9 Accessing IMS Via the OTMA Callable Interface	125
Functional Overview	126

What is the OTMA Callable Interface?	126
Programming for OTMA	127
Requirements	127
Sample Rules Provided	127
Session Termination	128
Usage Notes	129
z/OS and IMS System Requirements	129
Invoking the Tool and the System Map Table	129
Interpreter TCB Held for Communications	129
Error Handling	129
Example Rules and Tables	129
Chapter 10 Accessing External Routines	131
Functional Overview	132
How Does TIBCO Object Service Broker Process an External Routine?	132
Steps Required to Use an External Routine	132
Transaction Level of the Routine	133
Cleanup of System Service Requests	133
Error Handling	133
Observing Standard Conventions	134
Information Available to an External Routine	134
Use of the AMODE and RMODE Attributes	134
Storage Requirements	135
Example Assembler Program	136
Making a COBOL Program Compatible with TIBCO Object Service Broker	138
Requirements	138
COBOL Run-Units	138
Link-Edit and Runtime Options	139
Syntax Mapping	140
Example COBOL Program	141
Making a PL/I Program Compatible with TIBCO Object Service Broker	143
Requirements	143
Link-Edit Options	143
Syntax Mapping	144
Example PL/I Program	144
Making a C Program Compatible with TIBCO Object Service Broker	146
Requirements	146
Syntax Mapping	147
Sample	148
Identifying Your External Routine to TIBCO Object Service Broker	152
Specify the Table Entries	152
Add an Entry in the ROUTINES Table	152

Adding an Entry to the ARGUMENTS Table.	155
Calling the Routine	157
Put the Routine in a Load Library	157
Call the Routine From TIBCO Object Service Broker	157
Chapter 11 Using User Builtin Routines	159
Functional Overview	160
What are User Builtin Routines?	160
What Are the Requirements for User Builtin Routines?	160
Programming Considerations	161
Acquiring and Releasing Storage	161
Using the \$SAVE Macro	161
\$SAVE Macro Storage Usage	162
Sample User Builtin Routines	163
Samples Available	163
USRSLEEP	163
Chapter 12 Using the Interface to TIBCO Enterprise Message Service	167
TIBCO Object Service Broker EMS Interface	168
Purpose of TIBCO Enterprise Message Service (EMS)	168
Overview of TIBCO Object Service Broker EMS Interface	168
Calling EMS	169
Shareable Tools Available	169
Argument Mapping	169
Error Handling	172
Configuration	173
Initializing the EMS Interface	173
Multi-threaded Environment	173
Code Page Support	173
Sample Applications	175
Rules Samples	175
Supported EMS Functions	177
Chapter 13 Using the TIBCO Service Gateway for WMQ	195
Overview	196
Usage Notes	196
Error Handling	197
Example Rule	197
Chapter 14 Introduction to the Call Level Interface	199
Aspects of the Call Level Interface	200

Purpose of the Call Level Interface	200
Supported Functionality	200
Supported Connections	201
Shared Addressing	201
Accessing Table Data Using the Host Languages Interface	201
Illustration of Generic User Client Using Call Level Interface	202
Functionality of the Call Level Interface	203
Start or Locate and Stop an Execution Environment	203
Start and Stop a TIBCO Object Service Broker Session	203
Start and End a TIBCO Object Service Broker Stream	204
Start and End a TIBCO Object Service Broker Transaction	204
Call a TIBCO Object Service Broker Rule	205
Finding the Name of a Rule in A Transaction	205
Operational Characteristics	206
Supported 3GL Languages	206
Multiple Execution Environments per Address Space	206
Standby Sessions	206
When Viewed by TIBCO Object Service Broker Administrator Tools	206
Call Level Interface Specification	207
What Is the Module to Call?	207
Example CALL Formats	207
Required Parameters	207
Usage of the Parameters	208
HRNHLLTM Module Parameters	209
Valid Call Parameters	209
Operational Parameters	210
Valid Calling Sequences	212
Calling Sequence	212
Permissible Transitions Between the Call Level Interface Functions	213
Examples of Typical Usage	214
Batch Client Example	214
Nested Execute Example	215
TRANSFERCALL Example	215
Using the Host Languages Interface	216
Writing a COBOL Program Using a Combination of the Call Level Interface, TIBCO Object Service Broker Access Statements, and SQL Statements	216
Additional Steps	217
Chapter 15 Preparing the Environment, Analyzing Returned Values, and Modifying Changes .	
219	
Preparing to Start or Locate the Execution Environment	220
Preparatory Steps	220

How to Analyze the Return and Reason Codes, and Returned Message	222
Evaluation of the Return and Reason Codes	222
Capturing the Returned Values	222
Examples	222
Call Level Interface Return Codes	225
Listing and Explanation	225
Call Level Interface Reason Codes	226
Listing and Explanation	226
Committing and Rolling Back Persistent Table Changes	231
Sample Calls	231
Returned Values	232
Chapter 16 Call Level Interface Functions.	233
Starting or Locating the Execution Environment – STARTEE	234
Syntax	234
Calling Parameters	234
Starting an Execution Environment	234
Obtaining Execution Environment Startup Parameters	234
Locating an Execution Environment	235
Returned Values	236
Advanced STARTEE Batch Usage	236
Stopping the Execution Environment – STOPEE	237
Syntax	237
Calling Parameters	237
Sample Calls	237
Returned Values	238
Starting the Session – STARTSS	239
Syntax	239
Calling Parameters	239
Sample Calls	239
Returned Values	240
Advanced STARTSS BATCH Usage	240
Stopping the Session – STOPSS	241
Syntax	241
Calling Parameters	241
Sample Code	241
Returned Values	242
Starting a Transaction – STARTTR	243
Syntax	243
Calling Parameters	243
What Limits the Number of Transactions	243
Sample Calls	244

Returned Values	244
Modifying Transactional Characteristics	245
Calling Parameters	245
What are the Transactional Characteristics?	245
What is the Inheritance of Transactional Characteristics?	245
Sample Calls	246
Returned Values	246
Ending a Transaction – STOPTR	247
Syntax	247
Calling Parameters	247
Sample Calls	247
Returned Values	248
Calling a Rule – CALLRULE	249
Syntax	249
Calling Parameters	249
DATA-IN and DATA-OUT Areas	249
Accessing the Storage Areas	251
Sample Calls	252
Return Values	253
Chapter 17 Multiple-Session Execution Environments in Batch	255
Starting Multiple-Session Execution Environments in Batch	256
What Facility Is Available?	256
Implementation Guidelines	256
Specifying an Environmental Wait Routine	257
Listing of the HRNXD Copybook	257
User Exit Types Supported	258
STARTEE Call	259
Behavior of STARTEE	259
Behavior in the Sample Programs	259
STARTSS Call	261
Purpose of STARTSS	261
Behavior in the Sample Programs	261
Sample Programs	262
Programs Provided	262
S6BEWTIN	262
S6BEWTSD	262
S6BEWTSS	263
COBCAPI3	264
Chapter 18 TIBCO Object Service Broker SDK (C/C++) Server	265
Introducing TIBCO Object Service Broker SDK (C/C++)	266

Required Parameters	266
Execution Environment Considerations	267
Preparatory Steps	267
Additional Requirements for CICS Execution Environments	269
SIT Parameter Requirements	269
Specifying the CICS Session Background Task Transaction	269
Specifying RACF Definitions	269
Chapter 19 Using TIBCO Object Service Broker SDK (C/C++)	271
Overview of the TIBCO Object Service Broker SDK (C/C++)	272
What Is the TIBCO Object Service Broker SDK (C/C++)?	272
How Does It Work?	272
How Can It Be Used?	272
Compiling and Running	273
Thread Safety	273
Constants	273
SDK (C/C++) Functions	275
cliProc	277
cliExecTran	289
cliSetCodepage	291
cliErrorReasonDescr	293
cliCommCreate	294
cliCommCreate1	294
cliCommDelete	295
cliCommFormat	295
cliCommFormat1	296
cliCommSegment	297
cliCommSegments	297
cliCommSegSize	298
cliCommSize	298
cliCommSizeCalc	299
cliCommSizeCalc1	299
LLCOPY_CSTR(listr, cstr)	300
LLCOPY_MEM(listr, prt, len)	300
LLDECLARE(name, len)	300
LLSETLEN(listr, len)	300
LLSTR(listr)	300
LLSTRLEN(listr)	300
Sample Application Using the SDK (C/C++)	301
C Program	301
Rule Called by Program	301
Table Referenced by a Rule	302
Output from the Program	302

- Chapter 20 Using TIBCO Object Service Broker SDK (Java) 303**
 - Overview of TIBCO Object Service Broker SDK (Java) 304
 - What Is the TIBCO Object Service Broker SDK (Java)? 304
 - Requirements 305
 - How Does It Work? 305
 - How Can It Be Used? 306
 - Compiling 306
 - Thread Safety 306
 - Constants 307
 - SDK (Java) Methods 308
 - Classes 308
 - Session Object Methods 312
 - Session 312
 - call 314
 - endMessage 317
 - execTran. 318
 - isActive. 320
 - reset. 320
 - shutdown 321
 - start 321
 - startTrans 323
 - stop 324
 - stopTrans 325
 - transNestLevel 325
 - userId. 326
 - SessionException Object Methods 327
 - SessionException 327
 - errorReasonDescr 328
 - reasonCode 328
 - rc 329
 - Misc Object Methods. 330
 - commCreate 330
 - commFormat 330
 - commSegmentInd 331
 - commSegments. 331
 - commSegSize 332
 - commSize 332
 - commSizeCalc. 333
 - readInt 333
 - readShort. 334
 - writeInt. 334
 - writeShort 335
 - Sample Application Using the SDK (Java) 336

Compiling and Running the Sample Program	336
Sample Rule Called by a Program	337
Sample Table Referenced by a Rule	337
Output from Program	338
Chapter 21 Coding TIBCO Object Service Broker Access Statements	339
Overview	340
How to Access TIBCO Object Service Broker Data	340
Steps Required	340
Samples Provided	341
Writing COBOL with TIBCO Object Service Broker Access Statements	342
Sample COBOL Program	342
Sample TIBCO Object Service Broker Table Definition	344
Coding the Access Statements	345
Where Do You Code the Access Statements?	345
Coding the Action Statements	345
Coding Considerations	346
Naming Differences Between TIBCO Object Service Broker and COBOL	346
How to Rename TIBCO Object Service Broker Names to Valid COBOL Names	346
Modifying the Join Character for Table.Field Names	347
Checking TIBCO Object Service Broker Runtime Errors	348
Coding Operators and Expressions	348
Embedding TIBCO Object Service Broker Action Statements	348
Chapter 22 Coding SQL Access Statements	351
Writing a COBOL Program with Embedded SQL Statements	352
Sample COBOL Program	352
Sample TIBCO Object Service Broker Table Definition	354
Coding SQL Access Statements	356
Initial Statement	356
Defining Valid Names	356
Specifying Selection	357
Specifying Data Areas	357
Coding the Remaining SQL Statements	357
Coding Considerations	358
Differences to Consider	358
Assigning Valid Names	358
Coding Operators and Expressions	359
Syntax Mapping	360
Error Checking and Handling	361
Error Checking	361
Error Handling Status Variables	361

Statements Supported by the SQL Preprocessor 363

 SQL Statements 363

 Supported Keywords and Clauses for the SELECT Statement 365

Chapter 23 Processing COBOL Programs 367

 Preprocessing the Access Statements 368

 Usage of HLIPREPROCESSOR 368

 Preparing the Program 370

 Steps Required 370

 Running the Program 372

 Steps Required 372

Appendix A SDK (C/C++) and SDK (Java) Error Reason Codes 373

 Listing of the Reason Codes 374

 Code Values and Explanations 374

Index 379

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 various external environments within a 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, and how to access programs written in external programming languages from within TIBCO Object Service Broker.

Topics

- [Related Documentation, page xviii](#)
- [Typographical Conventions, page xxiii](#)
- [Connecting with TIBCO Resources, page xxv](#)

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
code font	Code font identifies commands, code examples, filenames, pathnames, and output displayed in a command window. For example: Use <code>MyCommand</code> to start the foo process.
bold code font	Bold code font is used in the following ways: <ul style="list-style-type: none">• 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, <code>MyCommand</code> is enabled: <code>MyCommand [enable disable]</code>
<i>italic font</i>	Italic font is used in the following ways: <ul style="list-style-type: none">• To indicate a document title. For example: See <i>TIBCO ActiveMatrix BusinessWorks Concepts</i>.• To introduce new terms. For example: A portal page may contain several portlets. <i>Portlets</i> are mini-applications that run in a portal.• To indicate a variable in a command or code syntax that you must replace. For example: <code>MyCommand PathName</code>
Key combinations	Key name separated by a plus sign indicate keys pressed simultaneously. For example: <code>Ctrl+C</code> . Key names separated by a comma and space indicate keys pressed one after the other. For example: <code>Esc, Ctrl+Q</code> .
	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.

Table 1 General Typographical Conventions (Cont'd)


Convention	Use
	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
[]	<p>An optional item in a command or code syntax.</p> <p>For example:</p> <pre>MyCommand [optional_parameter] required_parameter</pre>
	<p>A logical OR that separates multiple items of which only one may be chosen.</p> <p>For example, you can select only one of the following parameters:</p> <pre>MyCommand param1 param2 param3</pre>
{ }	<p>A logical group of items in a command. Other syntax notations may appear within each logical group.</p> <p>For example, the following command requires two parameters, which can be either the pair param1 and param2, or the pair param3 and param4.</p> <pre>MyCommand {param1 param2} {param3 param4}</pre> <p>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:</p> <pre>MyCommand {param1 param2} {param3 param4}</pre> <p>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.</p> <pre>MyCommand param1 [param2] {param3 param4}</pre>

Connecting with TIBCO Resources

How to Join TIBCOCommunity

TIBCOCommunity is an online destination for TIBCO customers, partners, and resident experts, a place to share and access the collective experience of the TIBCO community. TIBCOCommunity 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 **Introduction**

This chapter describes the TIBCO Object Service Broker architecture, how to access TIBCO Object Service Broker from an external environment, and the main stages to setting up and processing within TIBCO Object Service Broker from within an external environment.

Topics

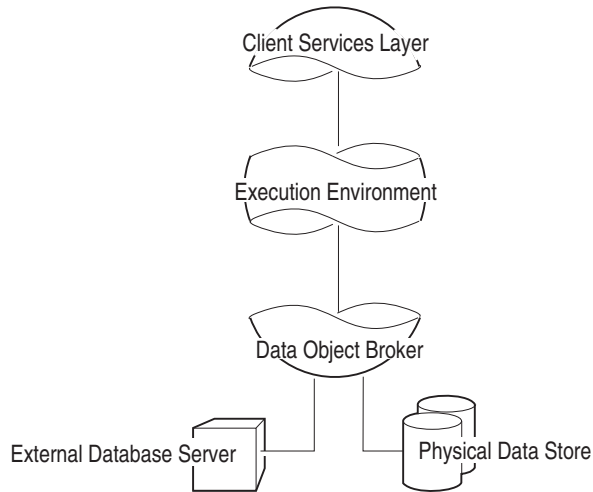
- [TIBCO Object Service Broker Architecture, page 2](#)
- [Accessing TIBCO Object Service Broker from an External Environment, page 5](#)
- [Stages to Setting Up and Processing Within TIBCO Object Service Broker, page 8](#)

TIBCO Object Service Broker Architecture

TIBCO Object Service Broker consists of three components:

- Client services layer
- Execution Environment
- Data Object Broker

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 as z/OS batch, TSO, CICS, IMS TM, or Native Execution Environment clients. Every TIBCO Object Service Broker session that runs in an Execution Environment is started by a client.

Execution Environment

The Execution Environment manages TIBCO Object Service Broker sessions, allowing you to execute rules and access tables and sessions. More than one Execution Environment can reside on a machine. An Execution Environment can reside on the same machine as a Data Object Broker or on a separate machine. Although the Execution Environment can interact with only one Data Object Broker directly, it can interact indirectly with other Data Object Brokers through distributed data access between Data Object Brokers.

The Execution Environment creates, manages, and terminates sessions. It establishes links with entities outside the Execution Environment. It delegates actions to a session once it is created and supervises the session while it exists.

The Execution Environment is established before any TIBCO Object Service Broker session is started and terminated after all sessions are done. Execution Environments are either single- or multiple-session.

TIBCO Object Service Broker Sessions

Within a session there can be one or more data access transactions. The session provides the context within which transactions can manipulate data and request access to relational data. Within this context:

- TIBCO Object Service Broker rules statements are executed.
- Requests to read from or write to tables are executed.
- The boundaries of data access transactions are defined and maintained.

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.

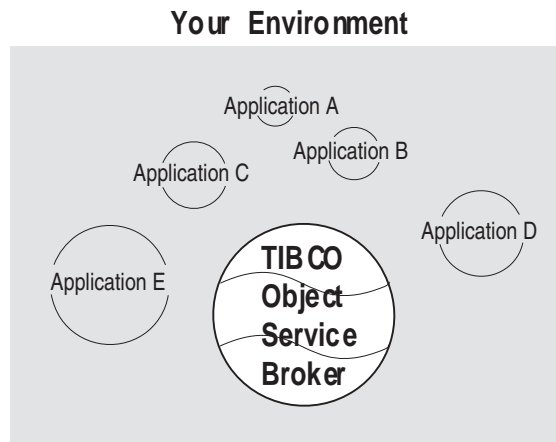
External Database Servers

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.

Accessing TIBCO Object Service Broker from an External Environment

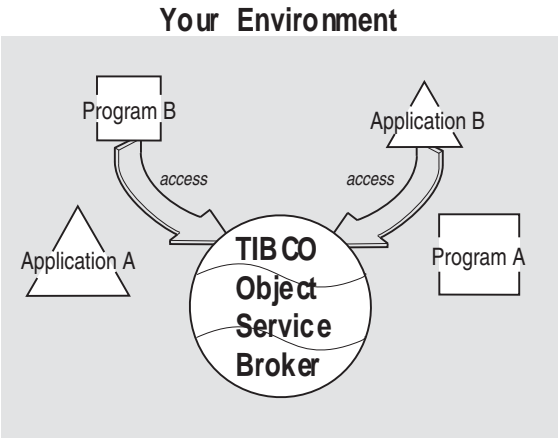
What is an External Environment?

All programs and applications run on an operating system platform, and often within a transaction manager such as CICS. In this manual these systems and transaction managers are referred to as *external environments*. Although applications can be written in different languages and have different areas of focus, they are sustained by, and commonly share the resources of, the environment where they run, as shown in the following diagram:



TIBCO Object Service Broker is Open to Its External Environments

In a similar way, TIBCO Object Service Broker makes its data and resources available to those applications that can make use of its Call Level Interface. Also, from within TIBCO Object Service Broker, users can access external routines and external data if they reside in the same address space as the session, as shown in the following diagram:



Facilities Available for Interfacing with External Environments

The following facilities provide access to and from the TIBCO Object Service Broker environment:

Facility	Refer to ...
TIBCO Object Service Broker clients	Chapter 4, TIBCO Object Service Broker Sessions Under z/OS Batch, page 31.
	Chapter 5, TIBCO Object Service Broker Sessions Under TSO, page 41.
	Chapter 6, TIBCO Object Service Broker Sessions Under the Native Execution Environment, page 51.
	Chapter 7, Using the TIBCO Service Gateway for CICS, page 59.
	Chapter 8, Using the TIBCO Service Gateway for IMS TM, page 97.

Facility	Refer to ...
External routine interface	Chapter 10, Accessing External Routines, page 131.
	Chapter 11, Using User Built-in Routines, page 159.
EMS Interface	Chapter 12, Using the Interface to TIBCO Enterprise Message Service, page 167.
TIBCO Service Gateway for WMQ	Chapter 13, Using the TIBCO Service Gateway for WMQ, page 195.
Call Level Interface	Chapter 14, Introduction to the Call Level Interface, page 199.
	Chapter 15, Preparing the Environment, Analyzing Returned Values, and Modifying Changes, page 219.
	Chapter 16, Call Level Interface Functions, page 233.
	Chapter 17, Multiple-Session Execution Environments in Batch, page 255.
TIBCO Object Service Broker SDK (C/C++)	Chapter 18, TIBCO Object Service Broker SDK (C/C++) Server, page 265.
	Chapter 19, Using TIBCO Object Service Broker SDK (C/C++), page 271.
TIBCO Object Service Broker SDK (Java)	Chapter 20, Using TIBCO Object Service Broker SDK (Java), page 303.
Host Languages Interface	Chapter 21, Coding TIBCO Object Service Broker Access Statements, page 339.
	Chapter 22, Coding SQL Access Statements, page 351.
	Chapter 23, Processing COBOL Programs, page 367.
Object Integration Gateway	<i>TIBCO Object Service Broker Object Integration Gateway</i>
Batch	<i>TIBCO Object Service Broker Programming in Rules</i>
TIBCO Object Service Broker 3270 Access Adapter	<i>TIBCO Object Service Broker Getting Started</i>

Stages to Setting Up and Processing Within TIBCO Object Service Broker

Main Stages

The following list outlines the main stages to setting up and processing within TIBCO Object Service Broker from within an external environment:

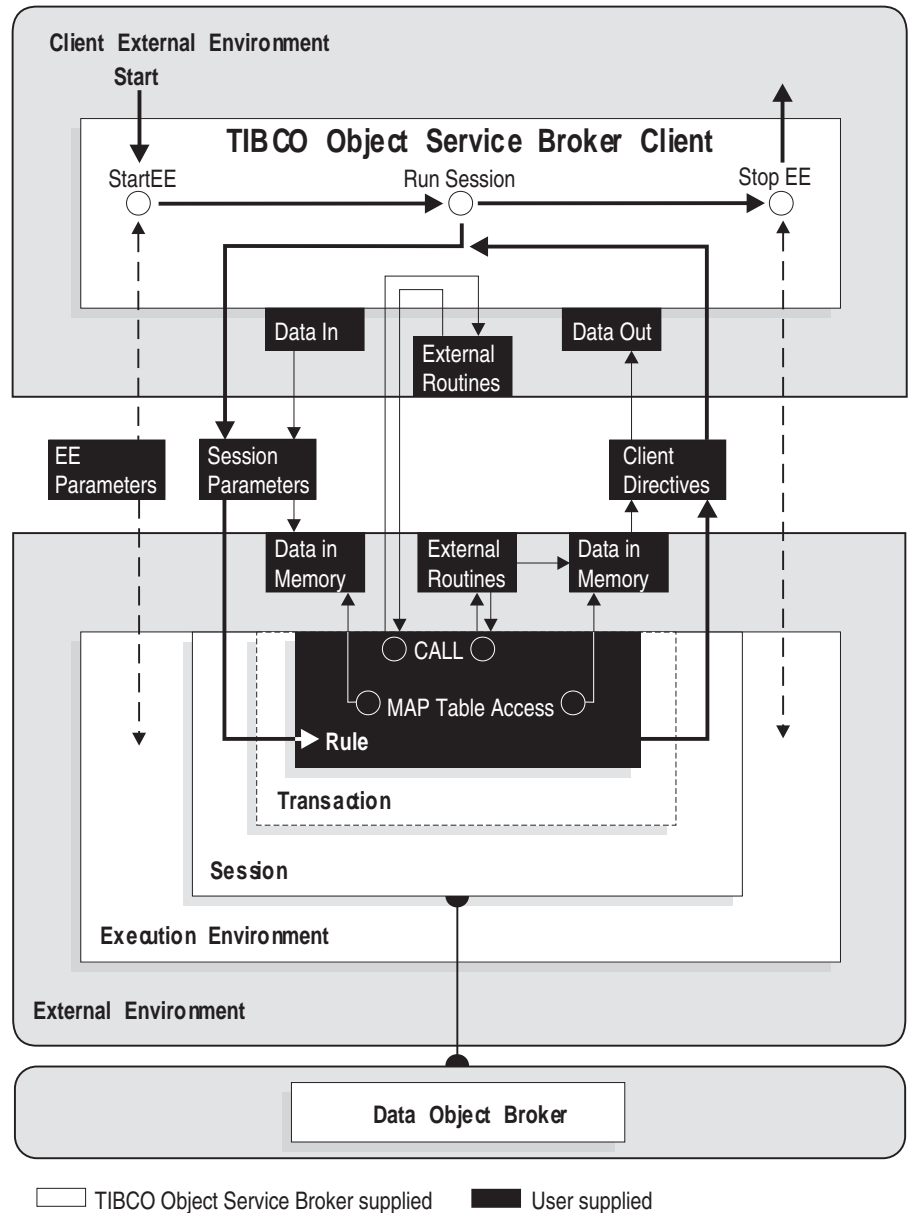
1. The user establishes an Execution Environment.
2. The user initiates a transaction using a client process.
3. The client starts a TIBCO Object Service Broker session. Depending on the client type, this step can be repeated for more than one session.
4. The Execution Environment gets environment and session parameters.
5. If TIBCO Object Service Broker security is in place, after connecting to the Data Object Broker, login security is performed.
6. The client activates a transaction. This transaction can start other transactions, call rules, and call external routines to manipulate data.
7. The sessions are terminated. As part of the termination, transactions are completed and resources cleaned up.
8. The client is notified that the session ended.
9. Data is returned to the external environment to notify it of the status of the session when the session ended.

See Also *TIBCO Object Service Broker Parameters* for more information about parameters.

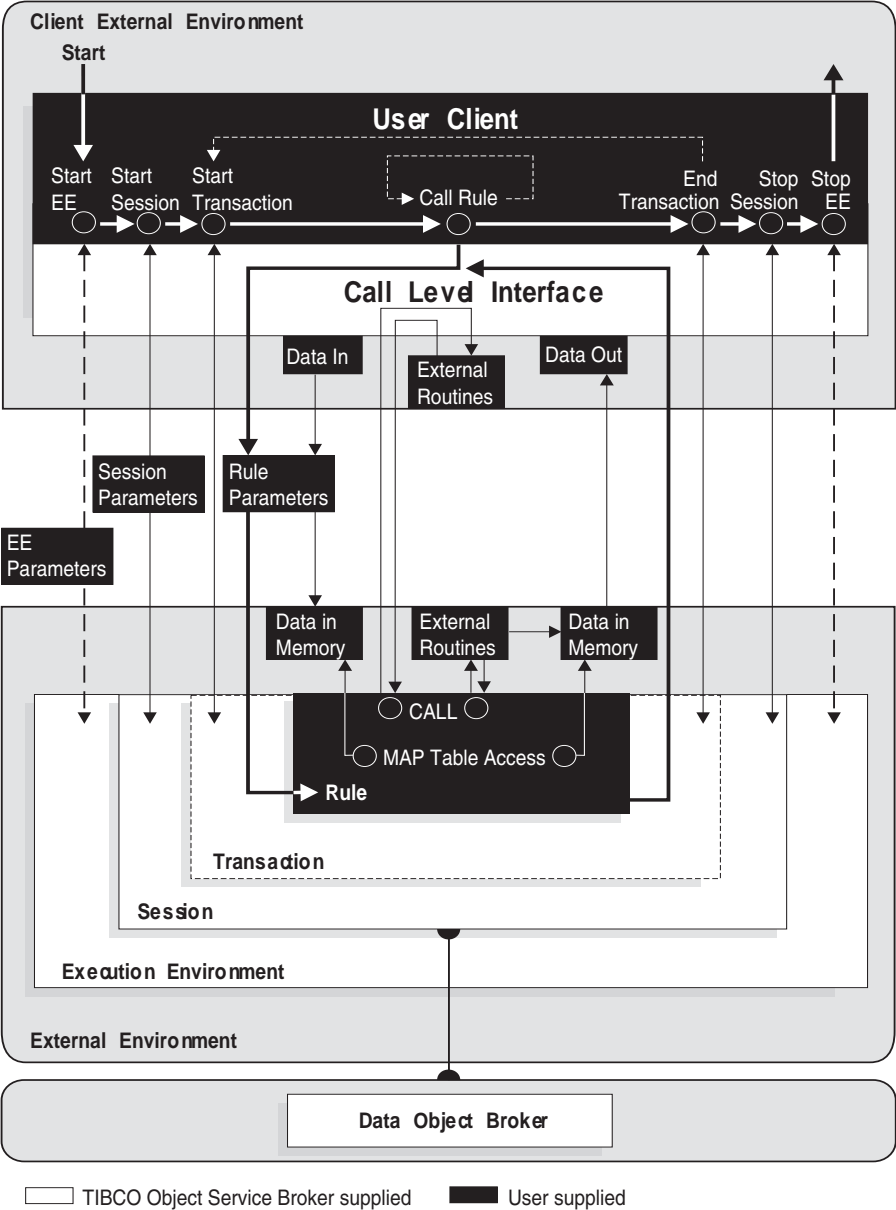
Interaction with TIBCO Object Service Broker and its External Environments

The following figures illustrate the stages described above in the context of the relationships between the Data Object Broker, the external environment, a TIBCO Object Service Broker client or a user client, the Execution Environment, the session, and the transaction. The details of these relationships are presented in the following chapters.

Data Flow When Using a TIBCO Object Service Broker Client



Data Flow When Using a User Client



Chapter 2

The TIBCO Object Service Broker Client Model

This chapter describes the TIBCO Object Service Broker client model, the different client styles, and how to set up the user profile for seamless clients.

Topics

- [Overview, page 12](#)
- [TIBCO Object Service Broker Client Styles, page 15](#)
- [Setting Up the User Profile for Seamless Clients, page 18](#)

Overview

Clients can either be:

- TIBCO Object Service Broker clients, supplied with the product
- User clients, written in COBOL, PL/1, C, or assembler as part of an existing application

The client environment you use determines whether:

- You can start one or more sessions from within your client process or transaction
- The client shares the same external environment as the Execution Environment
- The client and external environment share the same address space

TIBCO Object Service Broker Clients

Single-Session Clients

The following clients create and initialize a single-session Execution Environment, and start a session. When a user terminates the session, the Execution Environment is also terminated.

Batch client	<p>A batch client starts TIBCO Object Service Broker as a z/OS batch process.</p> <p>Batch clients are described in Chapter 4, TIBCO Object Service Broker Sessions Under z/OS Batch, on page 31.</p>
TSO client	<p>A TSO client starts TIBCO Object Service Broker as a z/OS TSO process. All TSO clients support a full-screen interface.</p> <p>TSO clients are described in Chapter 5, TIBCO Object Service Broker Sessions Under TSO, on page 41.</p>

Multiple-Session Clients

The following clients use multiple-session Execution Environments. These Execution Environments must be started before clients can start one or more sessions.

Native Execution Environment client	<p>You use a Native Execution Environment client to connect directly to TIBCO Object Service Broker using VTAM 3270. These clients support a full-screen interface. Sessions are started when the VTAM 3270 terminal is LOGON to the Native Execution Environment VTAM APPLID.</p> <p>Native Execution Environment clients are described in Chapter 6, TIBCO Object Service Broker Sessions Under the Native Execution Environment, on page 51.</p>
CICS client	<p>A CICS client starts a TIBCO Object Service Broker session within a CICS transaction. All CICS clients support a full-screen interface. CICS sessions are started when one of the CICS client programs is run. The CICS client and the session share the same address space. You can start the client as a conversational or pseudo-conversational CICS transaction or run it as a CICS program.</p> <p>CICS clients are described in Chapter 7, Using the TIBCO Service Gateway for CICS, on page 59.</p>
IMS TM client	<p>An IMS TM client starts a TIBCO Object Service Broker session using an IMS TM transaction. All TIBCO Object Service Broker IMS TM clients support a full-screen interface. IMS TM sessions are started in the Native Execution address space when one of the IMS TM programs is run in a Message Processing Region. You can start the client as a conversational or non-conversational IMS TM transaction.</p> <p>IMS TM clients are described in Chapter 8, Using the TIBCO Service Gateway for IMS TM, on page 97.</p>

User Clients

You can write your own user client, if you require a client to be written as part of a COBOL, PL/1, C, or assembler application. Use the Call Level Interface to create user clients for batch, TSO, and CICS environments. Refer to [Chapter 16, Call Level Interface Functions, on page 233](#) for detailed information.

TIBCO Object Service Broker SDK (C/C++) Client

The TIBCO Object Service Broker SDK (C/C++) client is an extension of the Call Level Interface. It extends the interface beyond the boundaries of the Execution Environment address space. It can use any communications protocol supported by TIBCO Object Service Broker.

The SDK (C/C++) client can reside in:

- Another address space in the same z/OS image
- Another z/OS system
- A platform other than z/OS

TIBCO Object Service Broker SDK (Java) Client

The TIBCO Object Service Broker SDK (Java) is an application programming interface (API) used by a Java application to manage TIBCO Object Service Broker sessions. It uses TCP/IP to connect to TIBCO Object Service Broker.

The SDK (Java) client can reside in any machine with a Java virtual environment.

See Also [Chapter 18, TIBCO Object Service Broker SDK \(C/C++\) Server, on page 265](#) for details about required setup steps in the Execution Environment for the SDK (C/C++) server

TIBCO Object Service Broker for z/OS Installing and Operating for more information about installing TIBCO Object Service Broker and its components.

Same Environments and Address Spaces

The external environment of the Execution Environment and the client are the same, and share the same address space for TIBCO Object Service Broker TSO, batch, and CICS clients, and all user clients using the Call Level Interface.

TIBCO Object Service Broker Client Styles

Clients connecting from CICS or IMS TM can make choices within four TIBCO Object Service Broker client styles:

- Seamless or non-seamless
- External user security or external transaction security
- Display or non-display
- Conversational or non-conversational

Default session attributes are applied to clients connecting from TSO, batch, and Native Execution Environment.

What Determines the Client Style?

The name of the client program determines the TIBCO Object Service Broker client style. Refer to the table in [Client Style Summary on page 17](#) for a listing of the program names and the types of styles for which they are used.

Seamless or Non-Seamless Client Styles

Seamless

To replace an existing non-TIBCO Object Service Broker transaction without changing other non-TIBCO Object Service Broker programs, use a seamless client program name. Seamless clients, by definition, cannot be passed a session parameter string. The external transaction name is used as the TIBCO Object Service Broker profile name. The startup rule named in the TIBCO Object Service Broker profile is the first rule run. Refer to [Setting Up the User Profile for Seamless Clients on page 18](#).

Non-Seamless

To provide session parameters not supported in the user profile or to explicitly specify the startup rule, use a non-seamless client program name. From the viewpoint of the external application, the session parameter string followed by application data is passed as data to TIBCO Object Service Broker.

Overriding the Default User ID

If the client style is non-seamless, the default user ID can be overridden by the USERID session parameter.

See Also *TIBCO Object Service Broker Parameters* for more information about parameters.

External User or External Transaction Security

External User Security

If you use an external user security program name, the default user ID of the TIBCO Object Service Broker sessions is the user ID authenticated by the external security manager. This ensures strict security on an individual user basis and requires that the external security manager authenticate the user ID.

External Transaction Security

If you use an external transaction security program name, the TIBCO Object Service Broker sessions uses the name of the external transaction as its TIBCO Object Service Broker user ID. The external security manager is responsible for authorizing the execution of this external transaction.

Minimizing TIBCO Object Service Broker User IDs

To avoid setting up a TIBCO Object Service Broker user ID for every external user ID, use an external transaction client style. This sets the user ID to the external transaction name.

Display or Non-Display Clients

Display

If you use a display client program name, the TIBCO Object Service Broker sessions can use the TIBCO Object Service Broker DISPLAY, UNTIL... DISPLAY, and DISPLAY & TRANSFERCALL rules language statements to present data to a screen.

Non-Display

If you use a non-display client program name, screen I/O is not supported for the TIBCO Object Service Broker sessions.

Conversational or Non-Conversational Clients

Conversational

If you use a conversational client program name, the TIBCO Object Service Broker session started under IMS TM has a Scratch Pad Area (SPA). Screen I/O is supported.

Non-Conversational

If you use a non-conversational client program name, the TIBCO Object Service Broker sessions started under IMS TM does not have a Scratch Pad Area (SPA). Screen I/O is supported but an IMS Physical Terminal Input Edit exit routine must be installed first.

Client Style Summary

The following table identifies the program name for a particular external environment and client style. For details, refer to the appropriate chapter for each client.

	CICS	IMS TM	Batch	TSO	VTAM LU2
Seamless	S6BCSxx1	S6BIMxx1			
Non-Seamless	S6BCSxx2	S6BIMxx2	S6BBATCH	S6BTSO	LOGON APPLID(x)DATA(x)
External User	S6BCSSxx	S6BIMSxx			
External Transaction	S6BCSTxx	S6BIMTxx			
Display	S6BCSxCx			S6BTSO	LOGON APPLID(x)DATA(x)
Non-Display	S6BCSxNx		S6BBATCH		
Conversational		S6BIMxCx			
Non-Conversational		S6BIMxNx			

See Also *TIBCO Object Service Broker Managing Security* for the evaluation of user IDs and passwords.

TIBCO Object Service Broker Programming in Rules for rules language statements.

Setting Up the User Profile for Seamless Clients

Every external transaction name that is to be used as a seamless TIBCO Object Service Broker client must be defined to TIBCO Object Service Broker as a user ID. At session startup, the seamless client uses the external transaction name to select the user profile information to be used during the session.

Setting up A User Profile

Use the SE security manager workbench option to access the User Profile option in the MANAGE USERS area to set the profile for an external transaction. At minimum, provide a value in the **Startup Rule** field. You can also provide values for the **Action**, **Search**, **Browse**, **Library**, and **Current Group** fields.

Guidelines for Development Environments

In a development environment, you could find the startup rule for a session in a local library (that is, it was not promoted to the installation library). If this is the case in your environment, the default library in the user profile should be set to the local library ¹.

If the user ID for the external environment is being used as the user ID for TIBCO Object Service Broker, the startup rule must exist in the rules library for this user ID or the library specified in the user profile. In TIBCO Object Service Broker, the user ID determines permissible accesses to objects such as rules libraries and also the clearance of data accesses (that is, reading the rule for execution). Therefore, a user ID must have VIEW_DEFN to access a rule definition and READ to access the library containing the rule.

See Also *TIBCO Object Service Broker Managing Security* about user profiles and logging in to TIBCO Object Service Broker.

TIBCO Object Service Broker Programming in Rules for the rules language and rules libraries.

-
1. The default login library is that of the user ID (that is, user ENV00 uses default library ENV00 during login processing). If the default library is not set, the rule must reside in the library of the environmental user (the login library).

Chapter 3

Setting Execution Environment Parameters

This chapter describes how to set the execution environment parameters.

Topics

- [Usage of the Execution Environment Parameters, page 20](#)
- [Determining Session Characteristics, page 21](#)
- [Available Execution Environment Parameters, page 25](#)
- [Specifying Session Parameters Using an Input File or a CLIST, page 27](#)
- [Reducing Session Resources, page 28](#)

Usage of the Execution Environment Parameters

Purpose

An Execution Environment must be started before you can start a session. The parameters listed in this chapter determine the characteristics of that environment and of the session. When your session starts, your Execution Environment parameter values are merged with the values of the Execution Environment parameters that you specified for your session, to determine the characteristics of your session.

Precedence of Values

Values provided as part of your session startup string take precedence over the Execution Environment values. Refer to [Determining Session Characteristics on page 21](#) for more detail about session startup values and the order of evaluation for startup.

See Also *TIBCO Object Service Broker Parameters* for detailed information about the Execution Environment parameters.

TIBCO Object Service Broker for z/OS Installing and Operating for detailed information about how to start batch, TSO, CICS, and Native Execution Environments.

[Chapter 14, Introduction to the Call Level Interface, on page 199](#) to [Chapter 17, Multiple-Session Execution Environments in Batch, on page 255](#) for information about starting Execution Environments using the Call Level Interface

Determining Session Characteristics

You can set session parameters in a number of ways, as described in the following sections. You can also configure your session to use either a pre-supplied workbench or to execute a user-supplied or an installation-supplied rule.

When your session is activated, session parameter and Execution Environment parameter values are merged to determine the characteristics of your session. How this data is merged is decided by the order of evaluation described in the two tables below.

Login Authentication Required

Before your session can be activated you must pass login authentication. This authentication is based on the user ID you are using to activate your session.

See Also *TIBCO Object Service Broker Parameters* for more information about parameters.

Where to Specify Parameters for Single-Session Clients

The following table shows where you can specify the session parameters for clients running single sessions. If the same parameter appears in more than one place, the parameter value of the highest priority specification is used.

Specified In	Priority	Single-Session z/OS Batch	z/OS TSO
Session startup string	Highest	Y ^a	— ^b
Session parameter input file		Y ^c	Y ^c
User Profile		Y ^{d, e}	Y ^{d, e}
Installation default		Y ^f	Y ^f
TIBCO Object Service Broker-supplied default	Lowest	Y ^g	Y ^g
Default module name		S6BDRCB0 ^h	S6BDRCT0 ^h

a. Specified on the EXEC card using PARM='session startup string'. Maximum 100 characters.

- b. The TIBCO Object Service Broker CLIST creates a temporary file allocated to DDname HRNIN to pass all parameters.
- c. Parameter input file must be allocated to DDname HRNIN.
- d. The User Profile is bypassed if the session parameter NOPROFILE is used.
- e. The profile of the user ID is always used.
- f. The installation default configuration module name is determined by the type of Execution Environment. The default module name can be overridden by specifying the Execution Environment CONFIGURATION parameter.
- g. Refer to *TIBCO Object Service Broker for z/OS Installing and Operating* for TIBCO Object Service Broker default values.
- h. The second last character represents the type of Execution Environment, i.e., B=Batch and N=Native. This information is displayed in the Execution Environment startup message.

Where to Specify Parameters for Multiple-Session Clients

The following table shows where you can specify the parameters for clients running multiple sessions. If the same parameter appears in more than one place, the parameter value of the highest priority specification is used.

Specified In	Priority	Multiple-Session		
		CICS	IMS TM	Native
Session startup string	Highest	Y ^a	Y ^b	Y ^c
User Profile		Y ^{d, e}	Y ^{d, e}	Y ^{d, f}
Execution Environment startup string		Y ^g	Y ^h	Y ^h
Execution Environment parameter input file		Y ⁱ	Y ⁱ	Y ⁱ
Installation default		Y ^j	Y ^{j, k}	Y ^j

Specified In	Priority	Multiple-Session		
		CICS	IMS TM	Native
TIBCO Object Service Broker supplied default	Lowest	Y ^l	Y ^l	Y ^l
Default module name		S6BDRCC0 ^m	S6BDRCI0 ^m	S6BDRCN0 ^m

- a. Specified on the command line in the HURN session startup string or in the parameter area of the COMMAREA passed to a non-seamless TIBCO Object Service Broker/CICS interface program. Cannot be specified to a seamless TIBCO Object Service Broker/CICS interface program.
- b. Specified on the command line in the S6BLOGON session startup string or in the parameter segment to a non-seamless TIBCO Object Service Broker IMS TM client program.
- c. Specified when logging in to the Native Execution Environment using the DATA('session startup string') operand. Maximum 64 characters.
- d. The User Profile is bypassed if the NOPROFILE session parameter is used.
- e. The external transaction profile is used for seamless clients, and the user profile is used for non-seamless clients.
- f. The profile of the user ID is always used.
- g. Specified in the command line HINT region startup string.
- h. Specified on the EXEC PGM=S6BDR00 card in PARM='region startup string'. Max: 100 characters.
- i. Parameter input file must be allocated to the HRNIN DDname.
- j. The installation default configuration module name is determined by the type of Execution Environment. The default module name can be overridden by specifying the CONFIGURATION Execution Environment parameter.
- k. IMS TM sessions run in a Native Execution Environment.
- l. Refer to *TIBCO Object Service Broker for z/OS Installing and Operating* for TIBCO Object Service Broker default values.
- m. The second last character represents the type of Execution Environment, i.e., C=CICS, I=IMS, and N=Native. This information is displayed in the Execution Environment startup message.

See Also *TIBCO Object Service Broker Parameters* for detailed information about session parameters.

TIBCO Object Service Broker for z/OS Installing and Operating for detailed information about creating default configuration modules.

TIBCO Object Service Broker Managing Security about TIBCO Object Service Broker security and logging in to TIBCO Object Service Broker.

Available Execution Environment Parameters

Parameters Specific to the Execution Environment

The parameters listed below determine the characteristics of your Execution Environment. Some of the parameters in [Parameters for Your Session on page 26](#) can also be used to set up your Execution Environment.

BLTINNUM	CICSHURONTRAN	CICSPSEUDOCONVERSE
CICSREGIONSIZE	CICSVSAMS SYNC	CLIMSGLENMAX
COMMITSIZE	CONFIGURATION	EXECHASHSIZE
EXECLOCALNAMESIZE	EXECSCOPE SIZE	IBMFLOAT, NOIBMFLOAT
IMSSCREENATTRIBU	IMSSCREENTRAN	IMSSCREENTRANNC
INSTLIBNUM	LOGONRULENAME	MDL
NOSMFDETAIL	PEERSERVERID	PEERSERVERNUM
REGIONTABLESIZE	REGIONTRACE	REGIONTRACESIZE
REGIONTRACESIZE	SECACL SIZE	SECADMIN SIZE
SECAUDITLOG	SECOBJSIZE	SECPACL SIZE
SECURITY	SECUSER SIZE	SERVERID
SERVERS	SERVERTYPE	SMF TYPE
SORTTEXTMEMMAX	SORTINTMEMMAX	SORTINTNUMMAX
SORTINTPAGESMAX	SORTPGM	SORTUNIT
STANDBYNUM	STATSBUF	TAMBMAX
TAMBMIN	TAMBSTS	TASKEXECNUM
TASKFILENUM	TASKINITNUM	TASKMISCNUM
TASKOPERNUM	TASKSMFNUM	TASKSORTNUM
TDS	TEMPPRIMARYCYL	TEMPSECONDARY

TEMPUNIT	TEMPUNITCOUNT	TIMEOUTLIMIT
TRANMAXNUM		

Parameters for Your Session

The parameters listed below determine the characteristics of your session.

ACTION	BROWSE, NOBROWSE	CHARSET
CLHOST	CLINODE	CLIPORT
DECIMALSEPARATOR	EENAME	ERRMESSAGESCREEN
EXECLOCALSIZE	EXECSTACKSIZE	FILEGDGSEARCH
INSTLIB	LIBRARY	MSGLOGMAX
OAIBLOCKFACTOR	ONLINE, OFFLINE	PASSWORD
PRINTCLASS	PRINTCOPY	PRINTDATASET
PRINTDEST	PRINTFCB	PRINTFORM
PRINTUCS	PRINTXWTR	PROFILE, NOPROFILE
PROMPT, NOPROMPT	RULE	SEARCH
SESSIONENDACTION	SESSIONENDVALUE	SESSIONFILEMAX
SESSIONMEMMAX	SMFDETAIL	SMFPERFORMANCE
SORTPRINT	SORTWORKFILESMAX	STAE, NOSTAE
SYSLIB	TEST, NOTEST	TRANMEMMAX
USERID	VARLDELIMITER	VARRDELIMITER

See Also *TIBCO Object Service Broker Managing Security* about user profiles and logging in to TIBCO Object Service Broker.

Specifying Session Parameters Using an Input File or a CLIST

You can specify additional TIBCO Object Service Broker parameters using the DD statement HRNIN for all client styles, or a CLIST for TSO clients. The following sections describe the use of an input file. For information about using a CLIST refer to [Using a CLIST to Invoke TIBCO Object Service Broker on page 47](#).

Format of the Input File

The data set associated with HRNIN can be sequential, partitioned, or instream. The format of HRNIN must be as follows:

- The data set must be either RECFM=VB or RECFM=FB with LRECL=80
- Columns 73 through 80 are reserved for card sequence numbers for RECFM=FB data sets
- Parameters can be separated by commas
- Spaces are ignored
- Quoted strings can span multiple lines
- Comments are denoted with an asterisk (*) in column 1, or delimited anywhere between parameters by /* */

Example of Instream Parameter List Using HRNIN in JCL

```
...
//STEP1 EXEC PGM=S6BBATCH,REGION=4M,
//          PARM=(OFFLINE,'U=USERID,TDS=VTAMID')          > USER AND VTAM IDS
//STEPLIB DD DSN=S6B01.HURON.LOAD,DISP=SHR                > SHOULD CUSTOMIZE
//HRNOUT DD SYSOUT=*
//HRNPRNT DD SYSOUT=*
//HRNIN DD *                                              > RULE AND PARAMETERS
RULE=HLIPREPROCESSOR('COBOL','HURON','HURON AND COBOL SOURCE',
'COBOL ONLY SOURCE','HLL.HLLLIST','ERRORSTOP'),MSGLOGMAX=4M,
CHARSET=CDNB /* HRNIN comment */
/*
```



If a parameter is specified in both HRNIN and the PARM statement, the value in the PARM statement takes precedence.

Reducing Session Resources

Bypassing the Workbench

Bypassing the workbench at session startup lets you reduce session resources. To bypass the workbench, run a rule at session startup:

- In seamless clients, the name of the rule to run is obtained from the user profile with the same name as the client. The rule is run without arguments. In your Security user profile you must also leave the **Menu** field blank and/or transfercall to the rule, that is, set the **Action** field to T.
- In non-seamless clients, the RULE session parameter is used to execute a rule at session startup. You can run this rule with arguments. Refer to *TIBCO Object Service Broker Parameters* for information about the RULE parameter.

Operational Characteristics

When the session starts, you do not see the usual workbench menu; the rule named in the **Startup Rule** field of the User profile for seamless clients or with the RULE parameter for non-seamless clients is run immediately. If the rule fails, the TIBCO Object Service Broker session ends with a traceback and control is returned to the external environment.

Trapping Errors

You can trap any recoverable errors using an exception handler in the rule.

Obtaining Additional Data

For CICS, IMS TM, and the Call Level Interface, you can use MAP tables or the [\\$GETENVCOMMAREA](#) tool to obtain additional data that was passed to the session.

Changing the Invocation Options

You can change the way to invoke a rule with the ACTION parameter or user profile default. Instead of the default EXECUTE, use CALL (C) or TRANSFERCALL (T):

EXECUTE	The rule starts as a child transaction to the TIBCO Object Service Broker login transaction.
TRANSFERCALL	The rule is invoked as a new, separate transaction.
CALL	The rule runs as part of the initial TIBCO Object Service Broker login transaction.

Change other invocation attributes with parameters or user profile defaults such as SEARCH, BROWSE/NOBROWSE, and TEST/NOTEST.

Non-seamless CICS Client

In a non-seamless CICS client you can invoke the HURN transaction with a session startup string as follows:

HURN R=ABC(3,OPEN)

In this example, the ABC rule is passed two arguments: the first argument is the number “3” and the second is the string “OPEN”.

- See Also
- TIBCO Object Service Broker Programming in Rules* for error handling.
 - TIBCO Object Service Broker Managing Data* about MAP tables.
 - TIBCO Object Service Broker Shareable Tools* about tools.
 - TIBCO Object Service Broker Managing Security* about user profiles.
 - TIBCO Object Service Broker Parameters* about parameters.

Chapter 4

TIBCO Object Service Broker Sessions Under z/OS Batch

This chapter describes how to run and set TIBCO Object Service Broker sessions under z/OS batch and how to manipulate data in batch client sessions

Topics

- [How to Run Batch Applications, page 32](#)
- [How to Set Session Parameters, page 36](#)
- [How to Manipulate Data in a TIBCO Object Service Broker Batch Client Session, page 39](#)

How to Run Batch Applications

Invocation

Use TIBCO Object Service Broker batch clients when you want to run batch TIBCO Object Service Broker applications under JES2 or JES3 z/OS. You typically invoke the batch application as a job step using JCL. The batch client can be either the TIBCO Object Service Broker supplied batch client program S6BBATCH, or a batch program you wrote using the TIBCO Object Service Broker Call Level Interface.

Using the TIBCO Object Service Broker Supplied Batch Client Program

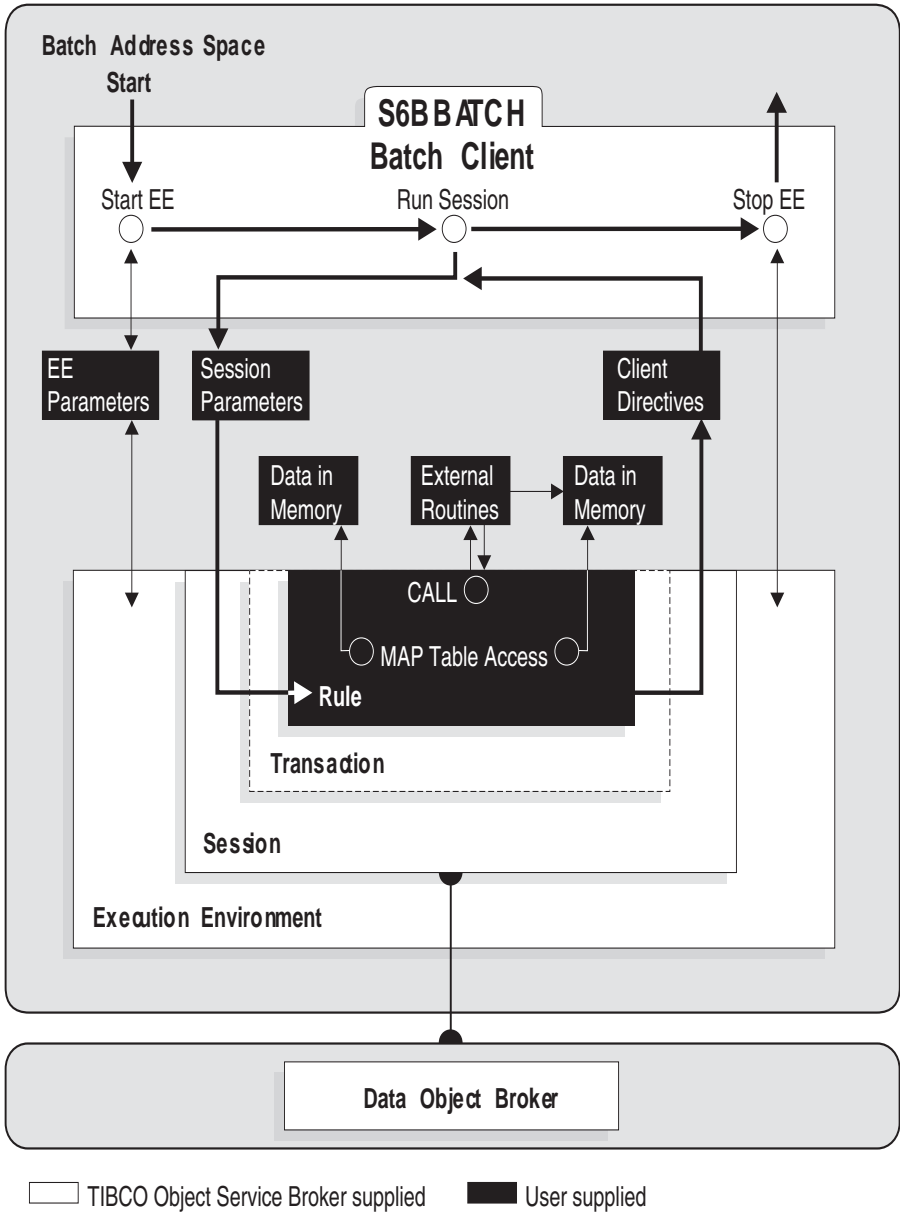
For a straightforward invocation of a batch application, use the S6BBATCH program supplied with TIBCO Object Service Broker. S6BBATCH starts a same-address space Execution Environment and runs the application as determined by the startup rule in the session.

When the application ends, the following events take place:

1. The session is terminated.
2. The Execution Environment is stopped.
3. A return code is passed by S6BBATCH, which can be used to control the execution of subsequent job steps.

Batch Client and the z/OS Environment

The following illustration shows how the TIBCO Object Service Broker batch client program fits into the z/OS environment:

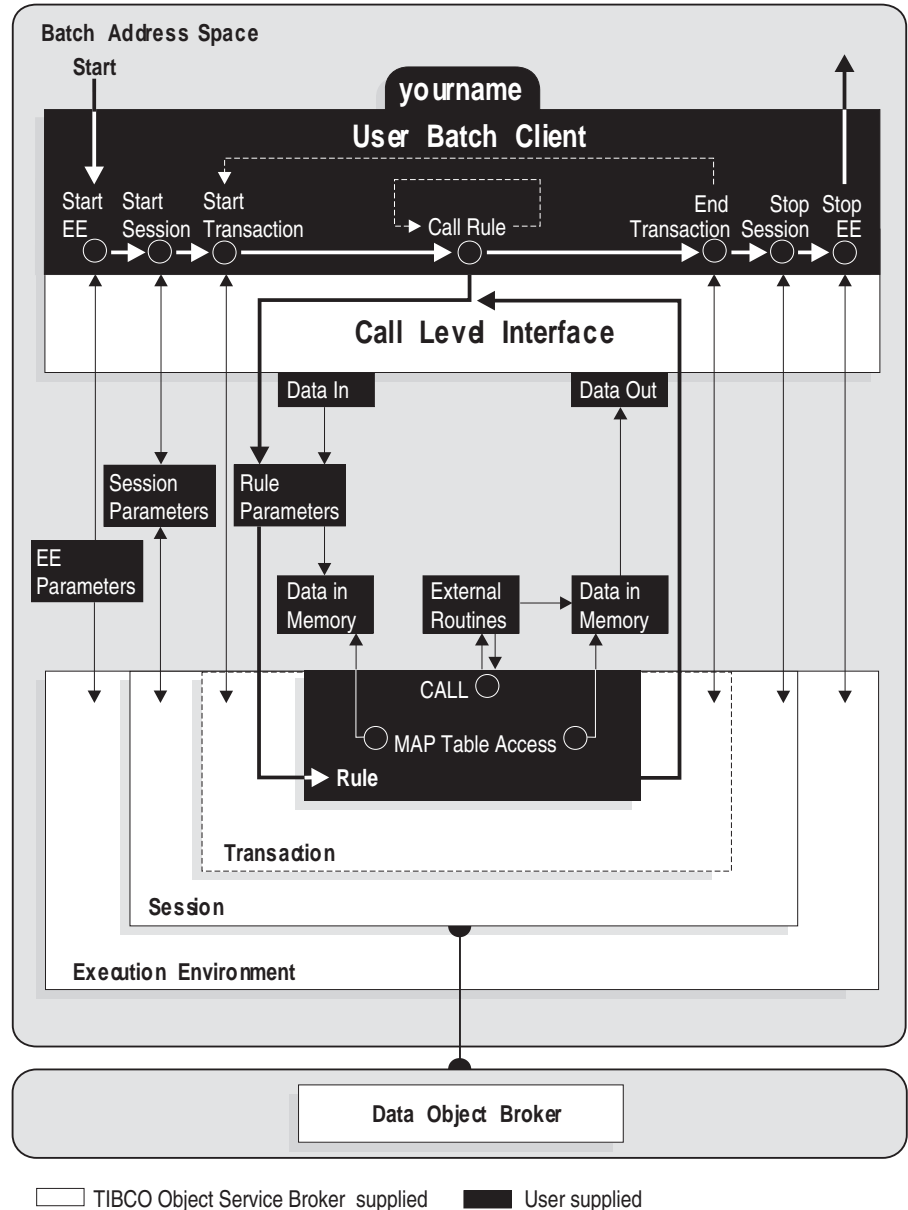


Using A Customized (User) Batch Client

If you want to access TIBCO Object Service Broker facilities from within a batch COBOL or assembler application program, you can write your own batch client using the TIBCO Object Service Broker Call Level Interface. In this case, your batch client program uses calls to the TIBCO Object Service Broker Call Level Interface to start a same address space Execution Environment, start a session, start a transaction, and run one or more rules in that transaction environment. Your batch client program is responsible for terminating the transaction, session, and Execution Environment before returning to z/OS.

User Batch Client and the z/OS Environment

The following illustration shows how your user batch client program fits into the z/OS environment. Refer to [Chapter 14, Introduction to the Call Level Interface](#), page 199 to [Chapter 17, Multiple-Session Execution Environments in Batch](#), page 255 for detailed information on how to write a batch application using the Call Level Interface.



How to Set Session Parameters

A number of options are available to you for determining the operational characteristics of your session. You can also require additional facilities to start a session. This section describes where to specify the options, the order of evaluation for these options, and the additional facilities you require.

Establishing Session Parameter Values

The following table describes where to specify the values for your session parameters and their order of evaluation, from highest to lowest:

Values Specified In...	Priority of Evaluation	Notes
Session startup string	Highest	For PGM=S6BBATCH: specify on the EXEC card using PARM='session startup string'. Maximum 100 characters. For TIBCO Object Service Broker Call Level Interface: specify as fourth parameter to HRNHLLTM to STARTSS operation. Refer to Starting the Session – STARTSS on page 239 .
Session parameter input file		The parameter input file must be allocated to DDname HRNIN.
User Profile		The profile of the TIBCO Object Service Broker user ID is used unless the session parameter NOPROFILE is specified.
Installation default		The installation default module is loaded from STEPLIB. The default configuration module name is S6BDRCB0, except when overridden by the CONFIGURATION Execution Environment parameter.
Default supplied by TIBCO Object Service Broker	Lowest	Refer to <i>TIBCO Object Service Broker for z/OS Installing and Operating</i> for default values.

Available DDnames

The DDnames listed in this table are used to run a batch Execution Environment:

DDname	Description
HRNEXTR	Optional partitioned data set containing load modules for user external routines.
HRNIN	Sequential file containing the Execution Environment and session parameters.
HRNLIB	Optional APF authorized partitioned data set containing the load modules required to run the Execution Environment when STEPLIB is not APF authorized.
HRNOUT	Sequential print file containing the system message log after an error, including error messages and rules tracebacks.
HRNPRNT	Sequential file containing print output generated by a rule with the PRT output medium, including reports, print tools, and TIBCO Object Service Broker tools such as PRINTTABLE .
STEPLIB	Partitioned data set containing the load modules required to run the Execution Environment.

See Also *TIBCO Object Service Broker for z/OS Installing and Operating* for more information on using HRNLIB to make sure that the TIBCO Object Service Broker load library is authorized.

Using an Instream Parameter List

The following is an example of an instream parameter list using HRNIN in JCL.

```
//MYJOB JOB ( , , ), USER=USERIDB
//* DISPLAY CLIENT STATUS
//*
//STEP1 EXEC PGM=S6BBATCH, PARM='TDS=S6BTEST', REGION=0M
//STEPLIB DD DSN=S6B.TST.LOAD, DISP=SHR
//HRNEXTR DD DSN=S6B.TST.LOAD, DISP=SHR
// DD DSN=MYLIB.LOAD, DISP=SHR
//HRNOUT DD SYSOUT=*
//HRNPRNT DD SYSOUT=*
//HRNIN DD *
        OFFLINE,
        U=USERIDA,
        P=USERPWD,
        TDS=S6BPROD,
        RULE=CLIENTSTATUS('X. SMITH')
/*
/*
```



The following takes place in this example:

- TDS=TEST in the PARM EXEC statement overrides TDS=PROD in HRNIN.
- The user ID and password are explicitly specified in HRNIN with the U and P session parameters, setting the TIBCO Object Service Broker user ID to USERIDA. If they are omitted, the user ID is the external ID specified in the USER parameter on the JOB card (that is, USERIDB).
- The OFFLINE parameter specifies that this is a non-conversational session. It cannot issue DISPLAY or DISPLAY & TRANSFERCALL statements.

See Also *TIBCO Object Service Broker Programming in Rules* about the rules language statements and writing rules.

TIBCO Object Service Broker Shareable Tools about the use of the tools.

TIBCO Object Service Broker Parameters about parameters.

How to Manipulate Data in a TIBCO Object Service Broker Batch Client Session

Passing Data to TIBCO Object Service Broker Batch Sessions

Batch clients can pass data to the session by:

- Placing data into the arguments of the rule to be called. Refer to [Calling a Rule – CALLRULE on page 249](#) for more information.
- Placing data into a block of storage pointed at by the DATA-IN area when a rule is called. Refer to [Calling a Rule – CALLRULE on page 249](#) for more information.
- Placing data in a table occurrence mapped by the Host Language Interface (HLI) and performing an INSERT or REPLACE. Refer to [Using the Host Languages Interface on page 216](#) for more information.

Returning from the Batch Client

Under normal circumstances, the batch client sets a return code of zero. You can use the `$SETSESSIONEND` tool to set the return code to a value between zero and 3999:

```
CALL $SETSESSIONEND( 'RC' , 8 );
```

or to cause a user abend:

```
CALL $SETSESSIONEND( 'ABEND' , 300 );
```

Determining the Next Step

You can subsequently use JES2 or JES3 condition processing to control the execution of subsequent job steps based on return or abend codes. For example:

```
//S1 EXEC PGM=S6BBATCH
.
//S2 EXEC PGM=S6BBATCH, COND=(EQ, 8)
.
```

Step S2 runs only if step S1 sets a return code other than 8.

Returning Data to a User Batch Client

Your called rule can return data to the user batch client by:

- Using MAP tables to REPLACE table occurrences in blocks of storage pointed at by the DATA-OUT area. Refer to [Calling a Rule – CALLRULE on page 249](#) for more information.
- Populating a temporary table and using the Host Language Interface to perform a FORALL or GET to the table containing the required data. Refer to [Using the Host Languages Interface on page 216](#) for more information.

Using External Routines

Your session can invoke external routines defined using the ROUTINES and ARGUMENTS tables to manipulate data. Refer to [Chapter 10, Accessing External Routines, on page 131](#) for more information.

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

TIBCO Object Service Broker Programming in Rules about the rules language statements and writing rules.

TIBCO Object Service Broker Shareable Tools about the use of the tools.

Chapter 5

TIBCO Object Service Broker Sessions Under TSO

This chapter describes how to run and set TIBCO Object Service Broker sessions under TSO and how to manipulate data in TSO client sessions.

Topics

- [How to Run TSO Applications, page 42](#)
- [How to Set Session Parameters, page 46](#)
- [How to Manipulate Data in TSO Client Sessions, page 49](#)

How to Run TSO Applications

Invocation

Use TIBCO Object Service Broker TSO clients to run TSO applications in the full-screen 3270 environment. You typically invoke the TSO client program via a tailored TSO CLIST. The TSO client program can be either the TIBCO Object Service Broker supplied program S6BTSO, or your own user TSO client program written using the TIBCO Object Service Broker Call Level Interface.

CLISTs are provided and are customized for your needs at installation time. You can choose to customize your CLIST to support more or fewer parameters. Refer to [Using a CLIST to Invoke TIBCO Object Service Broker on page 47](#). For the CLIST name and available parameters, see your system administrator.

Using the TIBCO Object Service Broker Supplied TSO Client Program

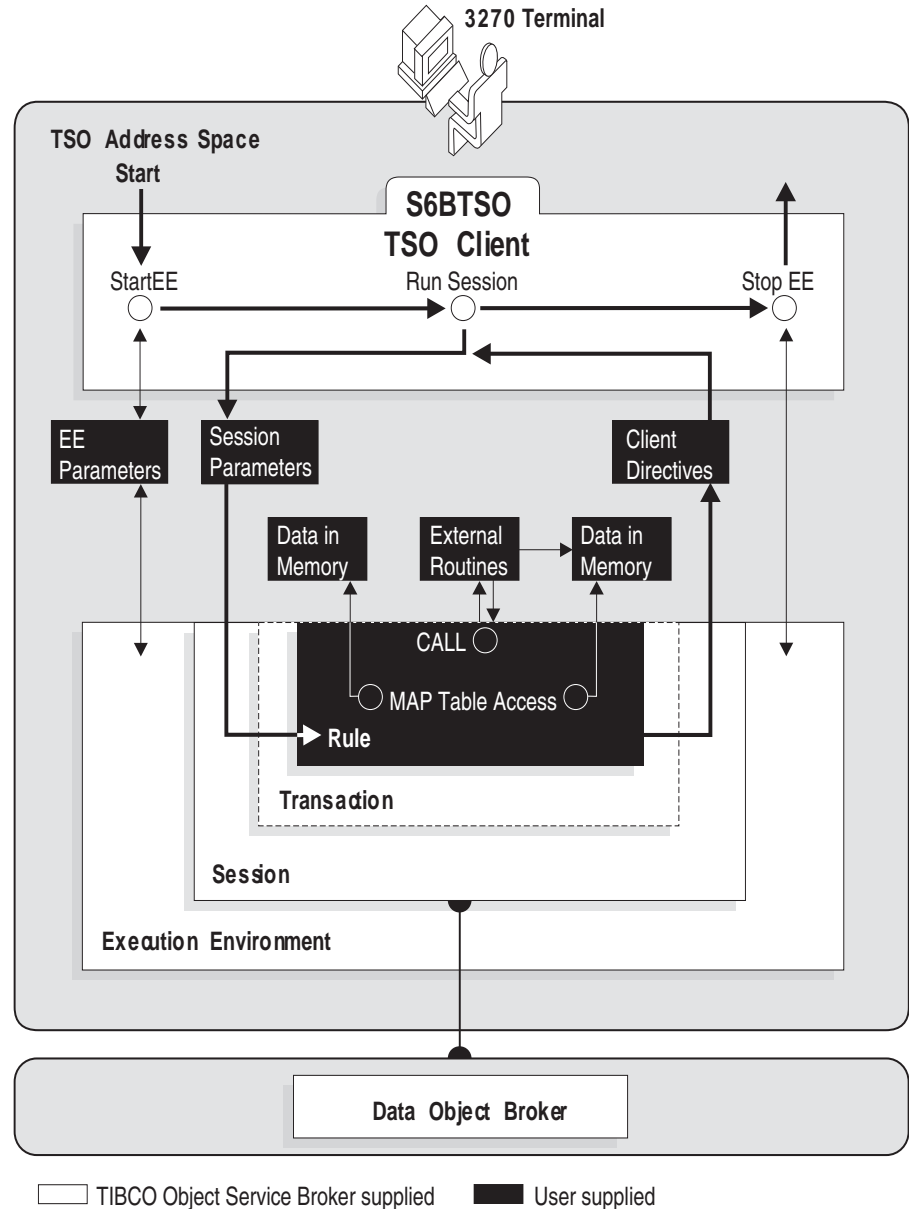
For a straightforward invocation of a TSO application, use the TIBCO Object Service Broker supplied S6BTSO program. S6BTSO first starts a same-address-space Execution Environment, and runs the TIBCO Object Service Broker application as determined by the start up rule in the session.

When the application ends, the following sequence of events occurs:

1. The session is terminated.
2. The Execution Environment is stopped.
3. A return code is returned by S6BTSO, which can be used to control the execution of subsequent TSO statements.

S6BTSO Starting a Single User Session

The following illustration shows how the S6BTSO program fits into the TSO environment:

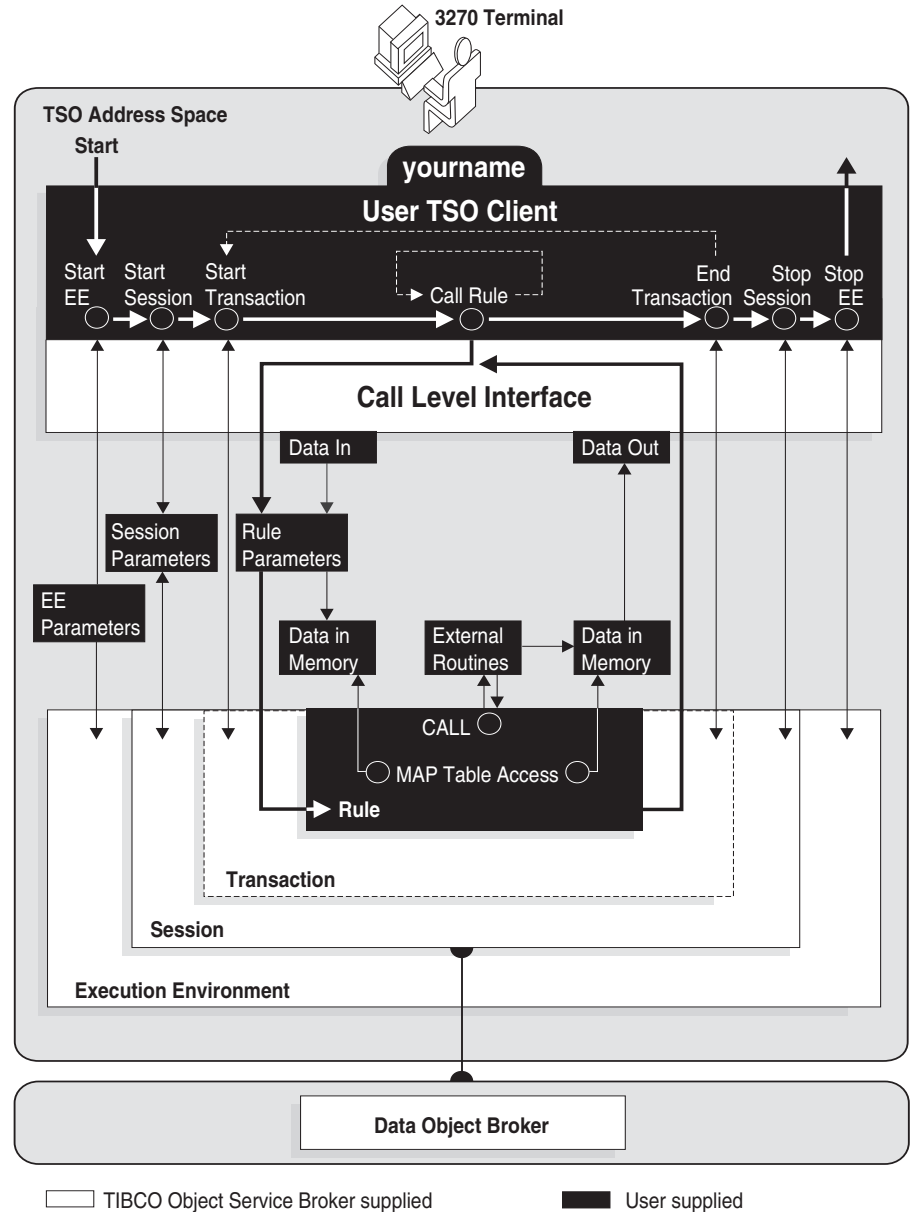


Using a Customized (User) TSO Client

To access TIBCO Object Service Broker facilities from within a TSO application program, you can write your own TSO client using the TIBCO Object Service Broker Call Level Interface. In this case, your application program calls to the Call Level Interface to start a same-address-space Execution Environment, start a session, start a transaction, and run one or more rules in that transaction. Your TSO application program is responsible for terminating the transaction, session, and Execution Environment before returning to z/OS.

User TSO Client and the z/OS Environment

The following illustration shows how your user TSO client runs a session in a TSO address space. Refer to [Chapter 14, Introduction to the Call Level Interface](#), page 199 to [Chapter 17, Multiple-Session Execution Environments in Batch](#), page 255 for detailed information on how to write a TSO application using the Call Level Interface.



How to Set Session Parameters

A number of options are available to you for determining the operational characteristics of your session. You can also require additional facilities to start a session. This section describes where to specify the options, the order of evaluation for these options, and the additional facilities you require.

Establishing Session Parameter Values

The following table describes where to specify the values for your session parameters and their order of evaluation, from highest to lowest:

Values Specified In ...	Priority of Evaluation	Notes
Session startup string	Highest	Supported when editing the USER CLIST only.
Session parameter input file		CLIST parameters are written to a parameter input file allocated to DDname HRNIN.
User Profile		The TIBCO Object Service Broker user profile is used unless the session parameter NOPROFILE is specified.
Installation default		The installation default module is loaded from the STEPLIB. Default configuration name is S6BDRCT0, except when overridden by the CONFIGURATION Execution Environment parameter.
TIBCO Object Service Broker default	Lowest	Refer to <i>TIBCO Object Service Broker for z/OS Installing and Operating</i> for TIBCO Object Service Broker-supplied default values.

Using a CLIST to Invoke TIBCO Object Service Broker

When you use parameters with your CLIST invocation of TIBCO Object Service Broker under TSO, the syntax is:

CLIST_NAME [{*parameter*[(*value*)]}]

where:

CLIST_NAME	Name of the CLIST that invokes TIBCO Object Service Broker.
<i>parameter</i>	The parameter name or abbreviation. Specify more than one parameter by leaving a space between each one.
<i>value</i>	The parameter value (if required) must be enclosed in parentheses.

You can use the USER CLIST to specify the Execution Environment load library and the external routine load library as follows:

LOADLIB (<i>libname</i>)	<i>libname</i> is the Execution Environment load library.
EXLIB (<i>libname</i>)	<i>libname</i> is the external routine load library allocated to DDname HRNEXTR. The programs in this library are written in a language other than the rules language and can be called by TIBCO Object Service Broker rules. You can specify or concatenate your own external library containing your own external routines. Refer to Chapter 14, Introduction to the Call Level Interface, page 199 to Chapter 17, Multiple-Session Execution Environments in Batch, page 255 for more information.

Example Invocation

The following example invokes a CLIST called **OSTAR**:

```
OSTAR NOBROWSE L(DEV00) RULE('TEST2(''A STRING'')')
EXLIB(USR30.OSTAR.ROUTINE)
```

In this example, the following takes place:

- This session starts with the parameters NOBROWSE (a parameter that does not require a value), and LIBRARY (abbreviated to L), which has a value of DEV00.
- The TEST2 startup rule is passed the text string: "A STRING".
- TIBCO Object Service Broker uses the data set USR30.OSTAR.ROUTINE to search for external routines.

If the USER CLIST does not provide the user ID to S6BTSO, the user ID is set to the TSO ID.

Specifying or Concatenating a Load Library

If your installation does not have a common external routine load library, or if you want to use your own load library, you can define your own library. To identify your external routine load library to TIBCO Object Service Broker, specify or concatenate a load library in the CLIST to invoke TIBCO Object Service Broker.

To specify a load library, provide its full name to the EXLIB parameter in the CLIST. For example:

```
EXLIB(USR30.OSTAR.ROUTINE)
```

In this example, TIBCO Object Service Broker searches the data set USR30.OSTAR.ROUTINE for external subroutines.

The order of the libraries determines the search order when you invoke an external routine, and the block size of the first library determines the block size for the others. If you cannot put the library with the largest block size first, use the BLKSIZE argument to specify a sufficient block size for all concatenated libraries.



Your Execution Environment should run authorized and your Data Object Broker must run authorized. (If you concatenate an authorized library with an unauthorized library, the resulting load library concatenation becomes unauthorized.)

See Also *TIBCO Object Service Broker for z/OS Installing and Operating* for information on using HRNLIB to make sure that the TIBCO Object Service Broker load library is authorized.

USER CLIST Distributed with TIBCO Object Service Broker

The USER CLIST, distributed with TIBCO Object Service Broker in the CLIST data set, runs a TIBCO Object Service Broker TSO client. The list of parameters that you can specify when you start your session—and the default values for some of the parameters—should be customized for your installation usage.

See Also *TIBCO Object Service Broker for z/OS Installing and Operating* about installation.
TIBCO Object Service Broker Programming in Rules about rules libraries and rules.
TIBCO Object Service Broker Shareable Tools about the use of the tools.
The TIBCO Object Service Broker Parameters about parameters.

How to Manipulate Data in TSO Client Sessions

Passing Data to TIBCO Object Service Broker TSO Sessions

User TSO clients can pass data to the session by placing data into any of these:

- The arguments of the rule to be called. Refer to [Calling a Rule – CALLRULE on page 249](#) for more information.
- A block of storage pointed at by the DATA-IN area when a rule is called. Refer to [Calling a Rule – CALLRULE on page 249](#) for more information.
- A table occurrence mapped by the Host Languages Interface and performing an INSERT or REPLACE. Refer to [Using the Host Languages Interface on page 216](#) for more information.

Returning from the TSO Client

Normally, S6BTSO sets a return code of 0. You can use the [\\$SETSESSIONEND](#) tool to set the return code to a value between 0 and 3999, for example:

```
CALL $SETSESSIONEND( ' RC ' , 8 )
```

or, to cause a user abend at session end:

```
CALL $SETSESSIONEND( ' ABEND ' , 300 )
```

The called rule can return data to the user TSO client by:

- Using MAP tables to REPLACE table occurrences in blocks of storage pointed at by the DATA-OUT area. Refer to [Calling a Rule – CALLRULE on page 249](#) for more information.
- Populating a temporary table and, from the user TSO client, using the Host Language Interface to perform a FORALL or GET on the table containing the required data. Refer to [Using the Host Languages Interface on page 216](#) for more information.

Using External Routines

Sessions can invoke external routines defined using the ROUTINES and ARGUMENTS tables. Refer to [Chapter 10, Accessing External Routines, on page 131](#).

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

TIBCO Object Service Broker Programming in Rules about the rules language statements and writing rules.

TIBCO Object Service Broker Shareable Tools about the use of the tools.

Chapter 6

TIBCO Object Service Broker Sessions Under the Native Execution Environment

This chapter describes how to run and set TIBCO Object Service Broker sessions under native execution environment and how to manipulate data in VTAM LU2 client sessions

Topics

- [Overview of the Native Execution Environment, page 52](#)
- [How to Set Session Parameters, page 54](#)
- [Manipulating Data in VTAM LU2 Client Sessions, page 57](#)

Overview of the Native Execution Environment

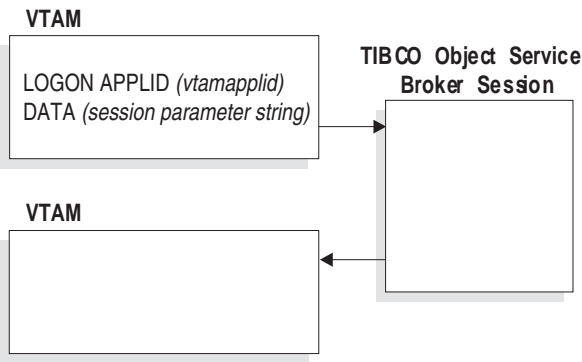
What is the Native Execution Environment?

The Native Execution Environment is a VTAM application that allows concurrent access to a Data Object Broker. Sessions are established by logging in through a VTAM LU2 (3270) terminal or from remote clients such as IMS TM clients running in an IMS TM Message Processing Region (MPR).

The Native Execution Environment is also optionally used to support remote peer servers for distributed data and external data servers such as TIBCO Service Gateway for Adabas, TIBCO Service Gateway for Datacom, and TIBCO Service Gateway for IMS/DB.

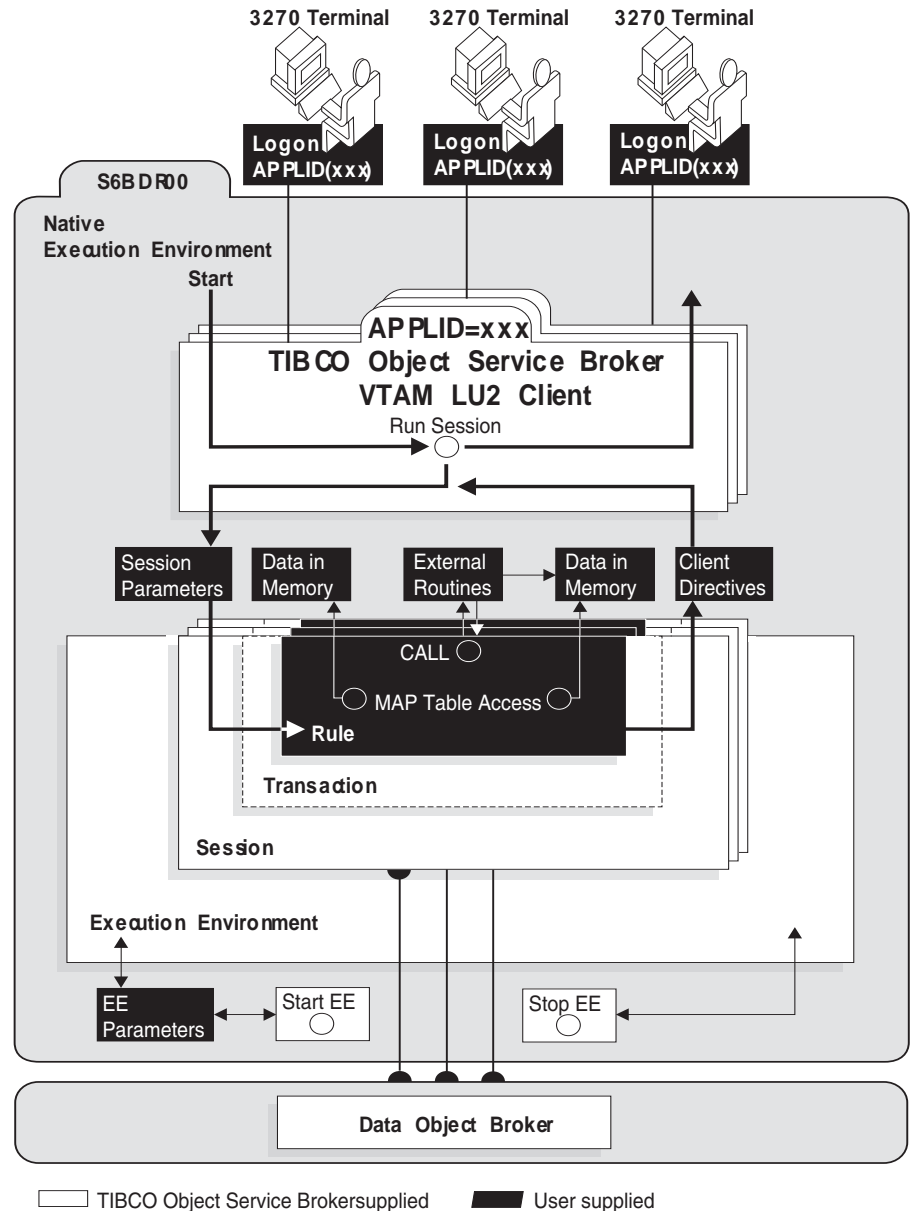
VTAM and TIBCO Object Service Broker Interaction

By logging on to the VTAM APPLID associated with a Native Execution Environment, you initiate a TIBCO Object Service Broker session. When the session ends, you return to VTAM or the network solicitor, as shown in the following figure:



3270 Terminal Session in a Native Execution Environment

The following figure shows how VTAM LU2 3270 terminals establish a session in a Native Execution Environment:



How to Set Session Parameters

The following sections describe where Execution Environment and session parameters are obtained and which DD names are required by the Native Execution Environment.

Where to Specify Session Parameters

The Native Execution Environment session defaults apply to all sessions running within it. The following table describes where to specify the values for your sessions and their order of evaluation, from highest to lowest:

Specified In	Priority	Notes
Execution Environment startup string	Highest	Specified on the EXEC PGM= S6BDR000 card in PARM= <i>'Execution Environment startup string'</i> .
Execution Environment parameter input file		Allocated to DDname HRNIN.
TIBCO Object Service Broker default	Lowest	Loaded from STEPLIB. Default configuration is S6BDRCN0 except when overridden by the CONFIGURATION parameter.



Certain parameters, such as SECURITY, cannot be specified in the HRNIN PARM=*'Execution Environment startup string'*. They must be added to the PARMNEE member in the TIBCO Object Service Broker CNTL data set, and followed by an EECONFIG assembly and link-edit job that creates a new S6BDRCN0 load module.

Other Applicable Sessions

In addition to providing session defaults for TIBCO Object Service Broker VTAM LU2 sessions, the session defaults also apply to TIBCO Object Service Broker IMS TM sessions, peer servers, and instances of TIBCO Service Gateway for Adabas, TIBCO Service Gateway for Datacom, and TIBCO Service Gateway for IMS/DB that are running in the Native Execution Environment. For further information about these additional topics, refer to [Chapter 8, Using the TIBCO Service Gateway for IMS TM, on page 97](#) or the appropriate *TIBCO Service Gateway*.

Available DDnames

The DDnames listed in the following table are used to run a Native Execution Environment:

DDname	Description
HRNEXTR	Optional partitioned data set containing load modules for user external routines.
HRNIN	Sequential file containing the Execution Environment and session parameter defaults.
HRNLIB	Optional APF authorized partitioned data set containing the load modules required to run the Execution Environment when STEPLIB is not APF authorized.
STEPLIB	The load library (partitioned data set) containing the load modules required to run the Execution Environment. Optionally, user external routines can be stored in this library.

See Also *TIBCO Object Service Broker for z/OS Installing and Operating* for more information on using HRNLIB to make sure that the TIBCO Object Service Broker load library is authorized.

Print Destination Restrictions

The following restrictions apply when specifying print destinations:

- Do *not* include the DDnames HRNOUT or HRNPRNT. These are session print files for single-session Execution Environments such as batch or TSO.
- Do *not* use the name S6BDRPRT as a print destination in a JES complex. This name is used by TIBCO Object Service Broker to run the SPOOLSTRIP job.

Establishing a TIBCO Object Service Broker VTAM LU2 Session

Communication is established with the Data Object Broker specified by the TDS parameter. Use the syntax below to log in to the Native Execution Environment that you require.

Syntax

```
LOGON APPLID(vtamapplid) DATA('U=userid,P=password')
```

where:

<i>vtamapplid</i>	The EENAME parameter value specified when the Native Execution Environment was initialized. Ask your system administrator to provide you with this value.
<i>userid</i>	A valid TIBCO Object Service Broker user ID that the Native Execution Environment uses to create a TIBCO Object Service Broker session.
<i>password</i>	The password of the user ID set by the U parameter can be optionally specified

Other session parameters can be specified in the DATA parameter, up to the VTAM restriction on the length of the DATA parameter string (64 characters).

See Also *TIBCO Object Service Broker for z/OS Installing and Operating* for additional information about installing and configuring the Native Execution Environment interface.

TIBCO Object Service Broker Parameters about parameters.

Manipulating Data in VTAM LU2 Client Sessions

Passing and Returning Data

No data can be passed to a TIBCO Object Service Broker VTAM LU2 session other than through arguments to the RULE parameter. It is forbidden to return data from the session.

Determining the Next Step

Upon termination of the TIBCO Object Service Broker VTAM LU2 session, the LU2 device is returned to the network solicitor.

Calling External Routines

Sessions running in the Native Execution Environment can invoke external routines using OS linkage defined in the ROUTINES and ARGUMENTS tables. Depending on the use, external routines can be accessed through TIBCO Object Service Broker VTAM LU2 sessions, TIBCO Object Service Broker IMS TM sessions, or peer servers. Refer to [Chapter 10, Accessing External Routines, on page 131](#) for more information.

This chapter describes how to run and set TIBCO Object Service Broker sessions under CICS, using the Service Gateway for CICS.

Topics

- [How to Run CICS Applications, page 60](#)
- [Session Initiation and Termination, page 64](#)
- [Selecting a TIBCO Object Service Broker CICS Client Program, page 66](#)
- [How to Set Session Parameters, page 68](#)
- [Starting TIBCO Object Service Broker Sessions, page 70](#)
- [Passing the COMMAREA Between a TIBCO Object Service Broker CICS Client and a Session, page 85](#)
- [How Can Data Be Returned, page 88](#)
- [Performing CICS Functions at Session End, page 90](#)
- [Calling External Routines, page 91](#)
- [CICS Channels and Containers in the TIBCO Object Service Broker CICS Session Environment, page 93](#)

How to Run CICS Applications

You can run CICS application using the Service Gateway for CICS, which enables you to use TIBCO Object Service Broker CICS clients to run z/OS CICS applications that access TIBCO Object Service Broker. The TIBCO Object Service Broker CICS clients can start as conversational or pseudo-conversational CICS transactions or you can run them as CICS programs.



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

CICS Client Programs

The CICS client program can be one of the eight TIBCO Object Service Broker CICS clients (S6BCS_{xxx}) or you can write your own user CICS client program using the TIBCO Object Service Broker Call Level Interface. Refer to [Selecting a TIBCO Object Service Broker CICS Client Program on page 66](#) and [Chapter 14, Introduction to the Call Level Interface, page 199](#) to [Chapter 17, Multiple-Session Execution Environments in Batch, page 255](#) for more information about these options.

Using the TIBCO Object Service Broker Supplied CICS Modules

For a straightforward invocation of a CICS client, use one of the S6BCS_{xxx} client programs (*xxx* can be any of eight different suffixes that represent different client programs). These are described in [Selecting a TIBCO Object Service Broker CICS Client Program on page 66](#). When invoked, the S6BCS_{xxx} module:

1. Locates the same-address-space Execution Environment.
Since CICS is a multi-user TP monitor, the Execution Environment is established separately, typically by the startup PLT.
2. Starts a session and passes it the contents of the CICS COMMAREA, which can include session parameters.

The session runs the application as determined by the startup rule.

At Application End

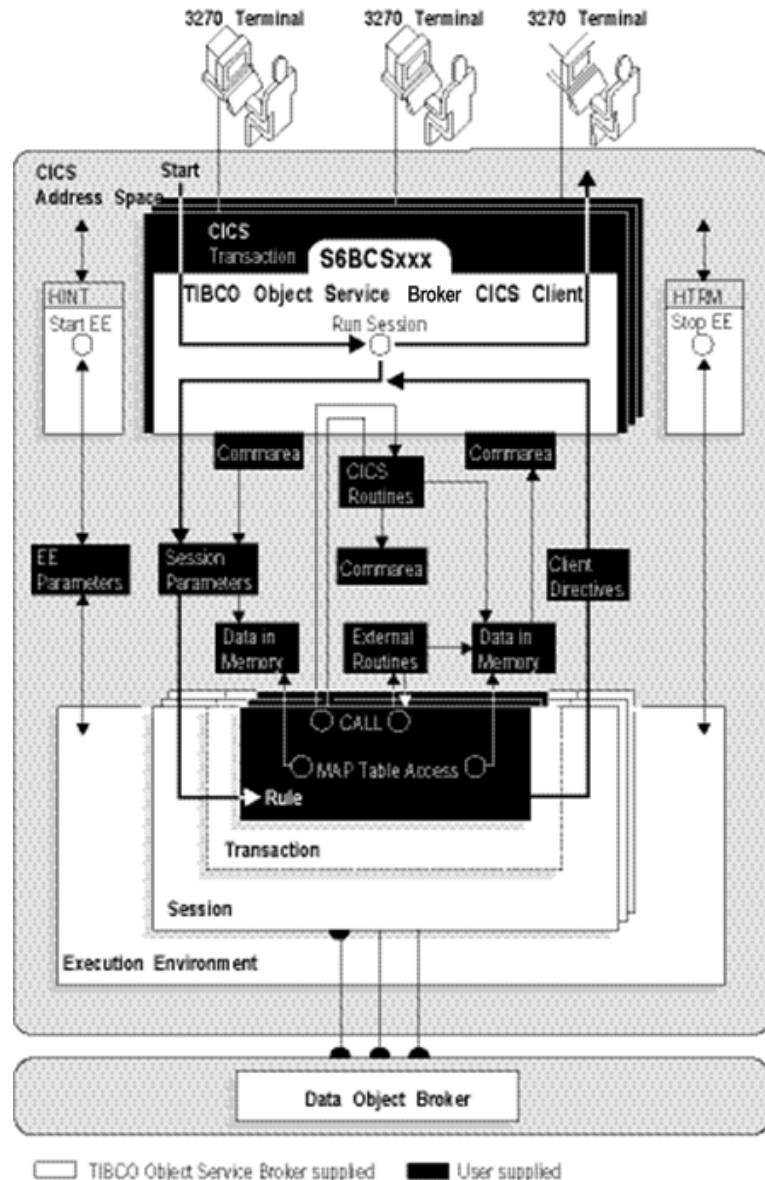
When the application ends:

1. It returns a possibly modified COMMAREA and CICS end session directives to the CICS client S6BCS_{xxx}.

- The CICS client program terminates using the CICS end session directives.

TIBCO Object Service Broker CICS Client Running a Session in a CICS Address Space

The following illustration shows how a typical TIBCO Object Service Broker CICS client (S6BCSxxx) runs a session in a CICS address space.



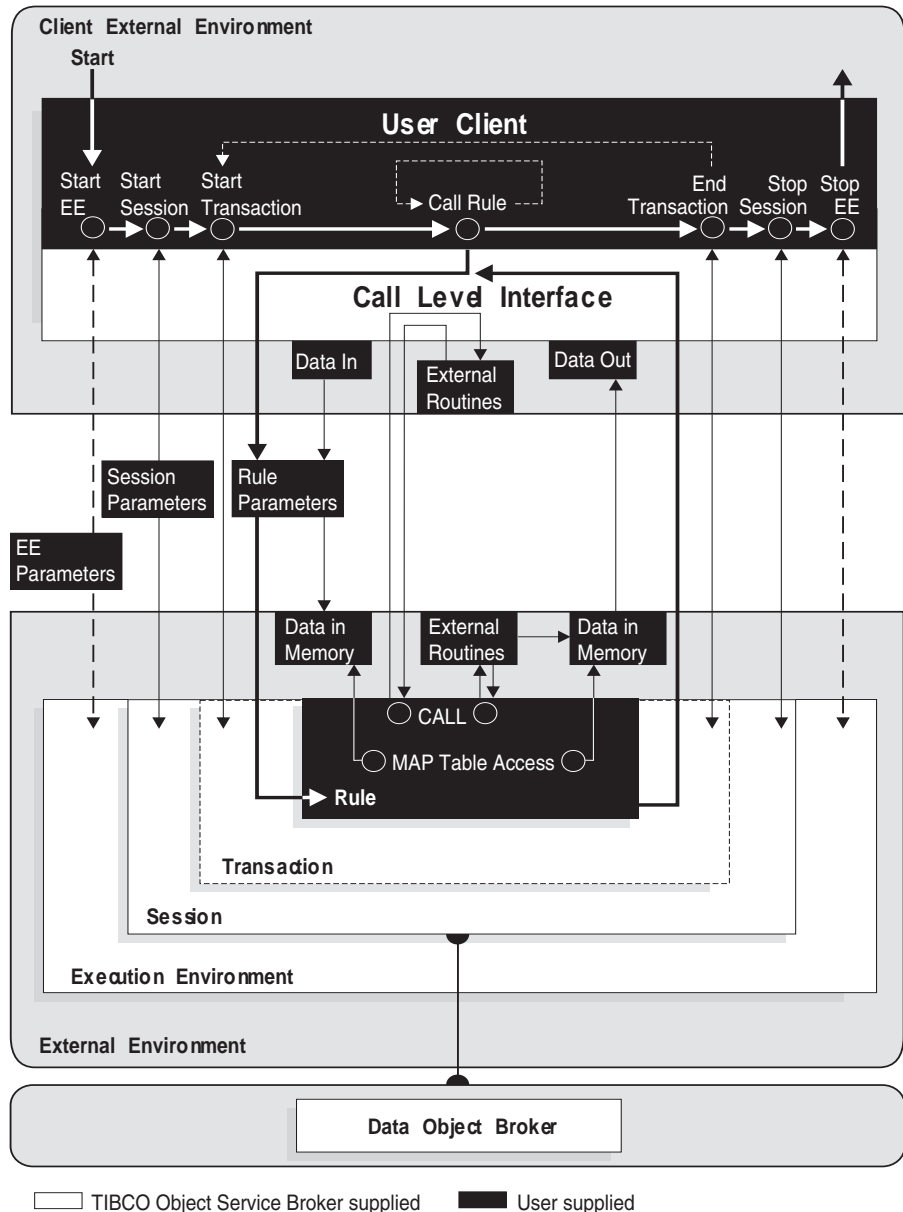
Using a Customized (User) CICS Client

To access TIBCO Object Service Broker facilities from within a CICS COBOL, PL/1, C, or assembler program, use the TIBCO Object Service Broker Call Level Interface to create your own user CICS client program. Your CICS client program:

- Uses calls to the TIBCO Object Service Broker Call Level Interface to locate the same-address-space Execution Environment to start a session, start a transaction, and run one or more rules in the transaction environment
- Is responsible for ending the transaction and stopping the session before terminating

A User CICS Client Runs a Session in a CICS Address Space

The following illustration shows how a typical user CICS client runs a session in CICS address space. Refer to [Chapter 14, Introduction to the Call Level Interface](#), page 199 to [Chapter 17, Multiple-Session Execution Environments in Batch](#), page 255 for detailed information about how to write a user CICS client program using the Call Level Interface.



Session Initiation and Termination

What Starts and Terminates an Execution Environment?

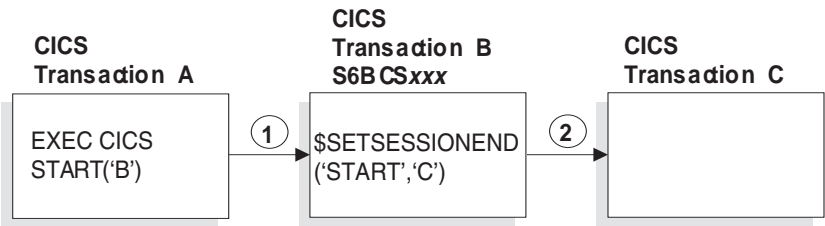
An Execution Environment must be started before a session can start. If the startup PLT was installed by the CICS administrator, the TIBCO Object Service Broker CICS Execution Environment is established at CICS region startup. You can also use the TIBCO Object Service Broker supplied **HINT** transaction to start the TIBCO Object Service Broker CICS Execution Environment. **HINT** accepts all Execution Environment and session parameters.

The Execution Environment is terminated at CICS region shutdown if the shutdown PLT is installed. You can also use the TIBCO Object Service Broker supplied **HTRM** transaction to terminate a TIBCO Object Service Broker CICS Execution Environment.

Methods of Session Initiation and Termination

When an Execution Environment is started, a CICS transaction can initiate a TIBCO Object Service Broker session and pass user data to it via the COMMAREA or Channel and Containers using one of the following methods:

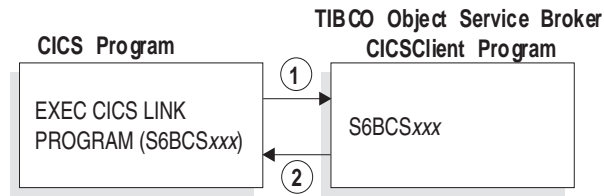
- **STARTing** a CICS transaction code associated with a TIBCO Object Service Broker CICS client program, as shown below:



- **XCTLing** to a TIBCO Object Service Broker CICS client program, as shown below:



- **LINKing** to a TIBCO Object Service Broker CICS client program, as shown below:



Replacing a CICS Transaction with TIBCO Object Service Broker Rules

Existing CICS transactions can be replaced by TIBCO Object Service Broker rules without changing interfaces to the previous and succeeding transactions.



Using these example programs, transaction A could START transaction B and transaction B could START transaction C. Initially, these could all be CICS transactions implemented as COBOL programs. Over time, the COBOL programs can be replaced by TIBCO Object Service Broker CICS clients that implement the same or enhanced functionality in TIBCO Object Service Broker rules. For example, in the first illustration, transaction B is replaced with a TIBCO Object Service Broker transaction.

See Also *TIBCO Object Service Broker for z/OS Installing and Operating* about the required PLTs and the available CICS transactions.

TIBCO Object Service Broker Parameters about parameters.

Selecting a TIBCO Object Service Broker CICS Client Program

You can choose from among eight different client styles. Each style corresponds to a different CICS program name. These programs are installed as part of the TIBCO Object Service Broker CICS interface.

TIBCO Object Service Broker CICS Client Programs

The following table lists the available client programs:

Transaction Name	Program Name	USERID set by	Display	Seamless
User supplied	S6BCSSC1	External user ID	Y	Y
HURN or user-supplied	S6BCSSC2	External user ID or USERID parameter	Y	N
User supplied	S6BCSSN1	External user ID	N	Y
User supplied	S6BCSSN2	External user ID or USERID parameter	N	N
User supplied	S6BCSTC1	CICS transaction name	Y	Y
User supplied	S6BCSTC2	CICS transaction name or USERID parameter	Y	N
User supplied	S6BCSTN1	CICS transaction name	N	Y
User supplied	S6BCSTN2	CICS transaction name or USERID parameter	N	N

Choosing the Right TIBCO Object Service Broker CICS Client Program

Match your style requirements with the appropriate TIBCO Object Service Broker CICS client program name in the following table:

S6BCSS _{xx}	TIBCO Object Service Broker user ID default is external user ID.
S6BCST _{xx}	TIBCO Object Service Broker user ID default is the CICS transaction name.
S6BCS _x C _x	Display client style, supports TIBCO Object Service Broker screen I/O.
S6BCS _x N _x	Non-display client style, does not support TIBCO Object Service Broker screen I/O.
S6BCS _{xx} 1	Seamless client, one data segment, uses transaction name to determine startup rule (refer to Seamless COMMAREA on page 86). ^a
S6CS _{xx} 2	Non-seamless client, first segment is the session parameter string, second segment is data. Use RULE= in session parameter string or TIBCO Object Service Broker user ID's profile to determine the first rule (refer to Non-Seamless COMMAREA on page 85). ^b

a. When using a non-display seamless client TIBCO Object Service Broker cannot prompt for a password; therefore, the Execution Environment SECURITY parameter cannot be set to INTERNAL or null.

b. Seamless clients, by definition, do not support session parameter overrides. The startup rule is obtained from the user profile associated with the transaction name. This rule is run with no arguments and with ACTION, SEARCH, BROWSE/NOBROWSE, and TEST/NOTEST parameters set based on user profile values or on the defaults. The rule obtains its data from the COMMAREA using MAP tables or the [\\$GETENVCOMMAREA](#) tool.

See Also *TIBCO Object Service Broker for z/OS Installing and Operating* about installing the CICS component of TIBCO Object Service Broker.

TIBCO Object Service Broker Parameters about parameters.

How to Set Session Parameters

A number of methods are available to you to set the operational characteristics for your session. You could also require additional facilities to start a session. This section describes these methods, their order of evaluation, and the additional facilities you could require.

Where to Specify CICS Execution Environment Parameters

The following table describes where to specify session parameter values and their order of evaluation, from highest to lowest:

Specified In	Priority	Notes
Execution Environment startup string	Highest	Not available if the Execution Environment is automatically started by the CICS startup PLT. All Execution Environment and session parameter defaults can be specified to the HINT transaction.
Execution Environment parameter input file		The parameter input file is allocated to DDname HRNIN.
Installation default	Lowest	The installation default module is loaded from the STEPLIB. The default configuration name is S6BDRCC0 except when overridden by the Execution Environment CONFIGURATION parameter. S6BDRCC0 is still used to determine the TIBCO Object Service Broker SVC number used to start the Execution Environment.

Available DDnames

You can use the following DDnames to run a CICS Execution Environment:

DDname	Description
DFHRPL	The external routine load library. It is used to load external routines if the external routine is defined to CICS as a CICS program. External routines that make CICS calls (EXEC CICS...) must be defined to CICS. Refer to Chapter 10, Accessing External Routines, on page 131 for more detail.
HRNEXTR	Optional partitioned data set containing load modules for user external routines. External routines not defined to CICS are loaded by TIBCO Object Service Broker from this library.
HRNIN	Sequential file containing the Execution Environment and session parameter defaults.
STEPLIB	The Execution Environment load library. The programs in this library run the Execution Environment and contain the default configuration module. In addition, external routines not defined to CICS are loaded from this library if the routines cannot be loaded from HRNEXTR.

Print Destination Restrictions

The following restrictions apply when specifying print destinations:

- Do *not* include the DDnames HRNOUT or HRNPRNT. These are session print files for single-session Execution Environments such as batch or TSO.
- Do *not* use the name S6BDRPRT as a print destination in a JES complex. This name is used by TIBCO Object Service Broker to run the SPOOLSTRIP job.

Synchronization of VSAM Files

You can control whether the VSAM server issues SYNCPOINTS under CICS using the CICSVSAMSYNC Execution Environment parameter. Refer to *TIBCO Object Service Broker Managing Data* for more information.

See Also *TIBCO Object Service Broker for z/OS Installing and Operating* for sample JCL needed to start a CICS region that supports an Execution Environment.

TIBCO Object Service Broker Parameters about the Execution Environment parameters.

Starting TIBCO Object Service Broker Sessions

Using the Command Line to Start a Session

To start a session from the CICS screen in command mode, enter a CICS transaction name (for example, **HURN**) that is associated with a TIBCO Object Service Broker CICS client program. Non-seamless client programs use screen input following the transaction name as the *session parameter string*. No data is passed to the TIBCO Object Service Broker session.

The TIBCO Object Service Broker transaction **HURN** uses the TIBCO Object Service Broker CICS client S6BCSSC2. S6BCSSC2 is non-seamless, permits screen I/O, and uses the CICS security user ID (which can be overridden with the USERID session parameter). When you use parameters with your **HURN** transaction, the syntax is:

HURN *parameter*[*=value*][*,*]

where:

<i>parameter</i>	The parameter name or abbreviation (valid abbreviations are listed in <i>TIBCO Object Service Broker Parameters</i>). Separate parameters on the command line with commas.
<i>value</i>	The parameter value (if any) follows an equal sign (=).

Error messages generated by TIBCO Object Service Broker or by the [ENDMSG](#) tool appear on your CICS screen.

Example of the HURN Transaction

An example of the **HURN** transaction is:

HURN U=USR09 ,PASSWORD=USR09

This establishes a session with the parameters USERID (abbreviated as U) and PASSWORD. You do not have to specify a user ID and password if you logged on to CICS using **CESN** or **CSSN** (or equivalent transactions). Since no rule is specified, the workbench appears, unless the profile of the user ID specifies a rule.

Using EXEC CICS START to Start a Session

To start a TIBCO Object Service Broker CICS client with **EXEC CICS START TRANSID**, do the following:


```

*MOVE LENGTH OF PARMS INTO COMMAREA
* # NOTE # THIS IS THE LENGTH OF THE PARM STRING ONLY
*           THE COMMAREA LENGTH MUST INCLUDE AT MINIMUM, THE PARM
*           LENGTH PLUS THE LENGTH OF THE PARM LENGTH FIELD ITSELF.
*
*           MVC    HURNPRML,=X'001B'
*
*MOVE PARMS INTO COMMAREA. RULE NAME AND PARMS SPECIFIED HERE ARE
*JUST EXAMPLES
*
*           MVC    HURNPARM,=C'RULE=TESTX(3245,BANK,DATES)'
*
*GET USER'S TERMINAL ID FOR SUBSEQUENT EXECUTION OF OSB-CICS
*AS STARTED TASK. FOR NON-TERMINAL STARTED TASK, THIS STEP CAN BE
*IGNORED.
*
*           EXEC   CICS ASSIGN                                X
*                   FACILITY(TERM)
*
*START OSB AS SEPARATE TASK AT USER'S TERMINAL USING DATA AREA
*TO PASS RULES AND PARMS.
*
*           EXEC   CICS START                                X
*                   TRANSID('HURN')                         X
*                   FROM(HURNPRML)                           X
*                   LENGTH(X'00A3')                           X
*                   TERMID(TERM)
*
* RETURN TO CICS
*
BREND      EXEC   CICS RETURN
           SPACE
           LTORG
BRTEXTL1  DC      Y(L'BRTEXT1)          FORCE OUT ASSEMBLY LITERALS
BRTEXT1   DC      C'BRTESTS - ABOUT TO  LENGTH OF LINK MESSAGE
           COPY   REGEQU
***>
*****>THE FOLLOWING DSECTS ALL EXIST IN CICS DYNAMIC STORAGE<*****
***>
*
* DSECT FOR BRTESTS
*
DFHEISTG DSECT
*
TERM      DS      CL4          TERMINAL ID
HURNCOMA  DS      0F           COMMAREA FOR STARTED TASK
HURNPRML  DS      H           LENGTH OF PARMS IN HURNPARM
HURNPARM  DS      CL161        PARMS (CAN BE LARGER)
HURNCOML  EQU     *-HURNCOMA    LENGTH OF DSECT
           SPACE
           END      BRTESTS

```

COBOL Example Using EXEC CICS START TRANSID

The following program shows **EXEC CICS START TRANSID**, which starts the TIBCO Object Service Broker CICS client program **S6BCSSC2** as a new transaction and does not return to the COBOL application.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.      CHISC2.
AUTHOR.          RICHARD PLANT
INSTALLATION.    CORPORATE.
DATE-WRITTEN.    5 APRIL 2007.

ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

    01 S6B-COMMAREA.
        02 S6B-PARM-AREA-LENGTH    PIC S9(4) COMP VALUE +40.
        02 S6B-PARM-AREA           PIC X(40) VALUE SPACES.
        02 S6B-DATA-AREA           PIC X(90) VALUE SPACES.

    01 S6B-COMMAREA-LENGTH         PIC S9(4) COMP VALUE +256.
    01 S6B-USERID                 PIC X(8)  VALUE SPACES.

LINKAGE SECTION.

    01 CICS-COMMAREA               PIC X(256).

PROCEDURE DIVISION USING CICS-COMMAREA.

    EXEC CICS ASSIGN USERID(S6B-USERID) END-EXEC.

    MOVE 'U=USR10'                 TO S6B-PARM-AREA.
    MOVE '== DATA AREA ==' TO S6B-DATA-AREA.
    MOVE S6B-COMMAREA TO CICS-COMMAREA.

    EXEC CICS TRANSID
        PROGRAM('S6BCSSC2')
        COMMAREA(CICS-COMMAREA)
        LENGTH(S6B-COMMAREA-LENGTH) END-EXEC.

    GOBACK.
```

Using EXEC CICS START to Start a Session with Channel

To start a TIBCO Object Service Broker CICS session with **EXEC CICS START TRANSID**, do the following:

1. Create the Channel and Container to be passed to the TIBCO Object Service Broker CICS Client.

For details, see [CICS Channels and Containers in the TIBCO Object Service Broker CICS Session Environment on page 93](#).

2. Execute the **CICS ASSIGN FACILITY** instruction to include the terminal ID in the **EXEC CICS START TRANSID** instruction.

Note: If you run TIBCO Object Service Broker as a background (nonterminal) task, skip this step.

3. **Optional.** For an additional level of security, execute the **EXEC CICS START TRANSID** instruction with different transaction IDs to start TIBCO Object Service Broker.

Assembler Example Using EXEC CICS START TRANSID with Channel

The following program shows **EXEC CICS START TRANSID**, which starts the TIBCO Object Service Broker CICS client program **S6BCSSC2** as a new transaction and does not return to the assembler application.

```

BRTESTSH TITLE 'OSB-CICS START TASK EXAMPLE PROGRAM'
* * * * *
*                               BRTESTSH                               *
*                               *                                       *
* FUNCTION: THIS PROGRAM SERVES AS AN EXAMPLE OF HOW OSB-CICS CAN    *
* BE ACCESSED FROM A USER APPLICATION PROGRAM. THIS                 *
* PROGRAM STARTS A OSB-CICS TRANSACTION AND PASSES                   *
* PARAMETERS TO OSB THROUGH A CHANNEL. THE MESSAGES                 *
* SENT FROM THIS PROGRAM INDICATE THE POINT AT WHICH THIS           *
* PROGRAM IS PROCESSING. FURTHER COMMENTS CAN BE FOUND              *
* AT EACH SECTION.                                                  *
*                               *                                       *
* LINKAGE:-                                                           *
*   STANDARD CICS.                                                    *
* * * * *
*                               SPACE                                   *
BRTESTSH DFHEIENT ,                                                  X
*                               CODEREG=R12
BRTESTSH CSECT
BRTESTSH AMODE 31
BRTESTSH RMODE ANY
*
BEGIN    DS      0H
*
* SEND MESSAGE INDICATING COMMENCEMENT
*
*                               EXEC CICS SEND                        X
*                               FROM(BRTEXT1)                        X
*                               LENGTH(BRTEXTL1)                     X
*                               ERASE                                  X

```

```

      WAIT
*
*   PUT Container SESSIONDATA for user session data
*
      MVC   STRSDATA,CC1SDATA      Copy session data
      LA     R0,L'STRSDATA         Get session data length
      ST     R0,STRSDATL           Set the length
*
      EXEC   CICS PUT CONTAINER('SESSIONDATA') CHANNEL('ABC')      X
            FROM(STRSDATA) FLENGTH(STRSDATL)                      X
            RESP(STRQRESP)
*
      MVC   STRERMSG,CC1MSG02      PUT container error message
      CLC   STRQRESP,DFHRESP(NORMAL) Command ok?
      BNE   CC1SENDM              No, send error message
*
*   PUT Container PARMCONTAINER for session parameters
*
      MVC   STRSPARM,CC1SPARM      Copy session parm
      LA     R0,L'CC1SPARM         Get session parm length
      ST     R0,STRSPRML           Set the length
*
      EXEC   CICS PUT CONTAINER('PARMCONTAINER') CHANNEL('ABC')    X
            FROM(STRSPARM) FLENGTH(STRSPRML)                      X
            RESP(STRQRESP)
*
      MVC   STRERMSG,CC1MSG02      PUT container error message
      CLC   STRQRESP,DFHRESP(NORMAL) Command ok?
      BNE   CC1SENDM              No, send error message
*
*GET USER'S TERMINAL ID FOR SUBSEQUENT EXECUTION OF OSB-CICS
*AS STARTED TASK. FOR NON-TERMINAL STARTED TASK, THIS STEP CAN BE
*IGNORED.
*
      EXEC   CICS ASSIGN                      X
            FACILITY(TERM)
*
*START OSB AS SEPARATE TASK AT USER'S TERMINAL USING DATA AREA
*TO PASS RULES AND PARMS.
*
      EXEC   CICS START TRANSID('HURN') CHANNEL('ABC')            X
            TERMID(TERM)
*
      MVC   STRERMSG,CC1MSG03      Trans HURN STARTed
      B     CC1SENDM
*
      Send message and return to CICS
*
CC1SENDM EXEC   CICS SEND FROM(STRERMSG) LENGTH(CC1ERMLN)
          EXEC   CICS RETURN
*
      LTORG ,                      FORCE OUT ASSEMBLY LITERALS
*
CC1SPARM DC     CL16'U=USR10,P=PUSR10'
CC1SDATA DC     CL36'This is user session data'
CC1ERMLN DC     Y(L'STRERMSG)
*

```

```

BRTEXTL1 DC      Y(L'BRTEXT1) LENGTH OF LINK MESSAGE
BRTEXT1  DC      C'BRTESTSH - ABOUT TO START OSB-CICS TASK'
CC1MSG02 DC      CL20'PUT CONTAINER error'
CC1MSG03 DC      CL20'Trans HURN STARTed'
*
      DFHREGS ,                      EQUate registers
***>
*****>THE FOLLOWING DSECTS ALL EXIST IN CICS DYNAMIC STORAGE<*****
***>
      DFHEISTG ,
*
TERM      DS      CL4
STRSDATL  DS      F
STRSDATA  DS      CL(L'CC1SDATA)
STRSPRML  DS      F
STRSPARM  DS      CL(L'CC1SPARM)
STRERMSG  DS      CL20
          END      BRTESTSH

```

COBOL Example Using EXEC CICS START TRANSID with Channel

The following program shows **EXEC CICS START TRANSID**, which starts the TIBCO Object Service Broker CICS client program **S6BCSSC2** as a new transaction and does not return to the COBOL application.

```

IDENTIFICATION DIVISION.
PROGRAM-ID.      CHISC2.
AUTHOR.          RICHARD PLANT
INSTALLATION.    CORPORATE.
DATE-WRITTEN.    8 DECEMBER 2010.

ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
    01 S6B-CONTAINER.
        02 S6B-SESSION-PARM-LENGTH PIC S9(5) COMP VALUE +16.
        02 S6B-SESSION-PARM        PIC X(16) VALUE SPACES.
        02 S6B-SESSION-DATA-LENGTH PIC S9(5) COMP VALUE +36.
        02 S6B-SESSION-DATA        PIC X(36) VALUE SPACES.
        02 TERM                    PIC X(4) VALUE SPACES.

PROCEDURE DIVISION.

    EXEC CICS ASSIGN FACILITY(TERM) END-EXEC.

    MOVE 'U=USR10,P=PUSR10' TO S6B-SESSION-PARM.

    EXEC CICS PUT CONTAINER('PARMCONTAINER') CHANNEL('ABC')
        FROM(S6B-SESSION-PARM) FLENGTH(S6B-SESSION-PARM-LENGTH)
        END-EXEC.

    MOVE 'This is user session data' TO S6B-SESSION-DATA.

```

```

EXEC CICS PUT CONTAINER('SESSIONDATA') CHANNEL('ABC')
      FROM(S6B-SESSION-DATA) FLENGTH(S6B-SESSION-DATA-LENGTH)
      END-EXEC.

EXEC CICS TRANSID('HURN') CHANNEL('ABC') TERMDID(TERM) END-EXEC.

GOBACK.

```

Using EXEC CICS LINK to Start a Session

To start a TIBCO Object Service Broker session using **EXEC CICS LINK**, establish a COMMAREA and LINK to the appropriate TIBCO Object Service Broker CICS client program.

The following program shows **EXEC CICS LINK**, which accesses TIBCO Object Service Broker CICS client program **S6BCSSC2** and returns to the assembler application.

```

BRTESTL TITLE      'OSB-CICS LINK EXAMPLE PROGRAM'
* * * * *
*                                     BRTESTL
*
* FUNCTION: THIS PROGRAM SERVES AS AN EXAMPLE OF HOW OSB-CICS CAN
*            BE ACCESSED FROM A USER APPLICATION PROGRAM. IT WILL
*            ASSEMBLE TO EXECUTABLE FORM. THIS PROGRAM LINKS TO
*            OSB-CICS BY PASSING PARAMETERS THROUGH A COMMAREA.
*            THE MESSAGES SENT FROM THIS PROGRAM INDICATE THE POINT
*            AT WHICH THIS PROGRAM IS PROCESSING. FURTHER COMMENTS
*            CAN BE FOUND AT EACH SECTION.
*
* LINKAGE:-
*   STANDARD CICS.
* * * * *
*            SPACE
BRTESTL  DFHEIENT,                                     X
          CODEREG=R12
BRTESTL  CSECT
BRTESTL  AMODE 31
BRTESTL  RMODE ANY
*
BEGIN    DS      0H
*
* SEND MESSAGE INDICATING COMMENCEMENT
*
          EXEC  CICS SEND                                X
                FROM(BRTEXT1)                            X
                LENGTH(BRTEXTL1)                          X
                ERASE                                       X
                WAIT
*
*MOVE LENGTH OF PARMS INTO COMMAREA
* # NOTE # THIS IS THE LENGTH OF THE PARM STRING ONLY.

```

```
*          THE COMMAREA LENGTH MUST INCLUDE AT MINIMUM, THE PARM
*          LENGTH PLUS THE LENGTH OF THE PARM LENGTH FIELD ITSELF.
*
MVC      HURNPRML,=X'001B'
*
*MOVE PARMS INTO COMMAREA. RULE NAME AND PARMS SPECIFIED HERE ARE
*JUST EXAMPLES.
*
MVC      HURNPARAM,=C'RULE=TESTX(3245,BANK,DATES) '
*
*LINK TO OSB CICS USING COMMAREA TO PASS RULES AND PARMS
*
EXEC     CICS LINK                                     X
          PROGRAM('S6BCSSC2')                          X
          COMMAREA(HURNCOMA)                            X
          LENGTH(X'00A3')
*
*SEND MESSAGE INDICATING LINK RETURN
*
EXEC     CICS SEND                                     X
          FROM(BRTEXT2)                                  X
          LENGTH(BRTEXTL2)                               X
          ERASE                                           X
          WAIT
*
* RETURN TO CICS
*
BREND    EXEC CICS RETURN
          SPACE
BRTEXTL1 DC  Y(L'BRTEXT1)                               LENGTH OF LINK MESSAGE
BRTEXT1  DC  C'BRTESTL - ABOUT TO LINK TO OSB/CICS'
BRTEXTL2 DC  Y(L'BRTEXT2)                               LENGTH OF RETURN MESSAGE
BRTEXT2  DC  C'BRTESTL- RETURNED SUCCESSFULLY FROM OSB/CICS'
          LTORG                                           FORCE OUT ASSEMBLY LITERALS
          COPY REGEQU
***> <***
*****> THE FOLLOWING DSECTS ALL EXIST IN CICS DYNAMIC STORAGE <*****
***> <***
*
* DSECT FOR BRTESTL
*
DFHEISTG DSECT
*
TERM      DS          CL4          TERMINAL ID
HURNCOMA  DS          0F          COMMAREA FOR LINK
HURNPRML  DS          H          LENGTH OF PARTMS IN HURNPARAM
HURNPARAM DS          CL161       PARMS (CAN BE LARGER)
HURNCOML  EQU          *-HURNCOMA  LENGTH OF DSECT
          SPACE
          END          BRTESTL
```


Using EXEC CICS LINK to Start a Session with Channel

To start a TIBCO Object Service Broker session with **EXEC CICS LINK**, create a Channel with the desired Containers and **LINK** to the appropriate TIBCO Object Service Broker CICS client program.

The following program shows **EXEC CICS LINK**, which accesses TIBCO Object Service Broker CICS client program **S6BCSSC2** and returns to the assembler application.

```

BRTESTLH TITLE 'OSB-CICS LINK EXAMPLE PROGRAM'
* * * * *
*                                     BRTESTLH                                     *
*
* FUNCTION: THIS PROGRAM SERVES AS AN EXAMPLE OF HOW OSB-CICS CAN
*           BE ACCESSED FROM A USER APPLICATION PROGRAM. THIS
*           PROGRAM LINKS TO OSB-CICS BY PASSING PARAMETERS THROUGH
*           A CHANNEL WITH CONTAINERS. THE MESSAGES SENT FROM THIS
*           PROGRAM INDICATES THE POINT AT WHICH THIS PROGRAM IS
*           PROCESSING. FURTHER COMMENTS CAN BE FOUND AT EACH
*           SECTION.
*
* LINKAGE:-
*   STANDARD CICS.
* * * * *
*           SPACE
BRTESTLH DFHEIENT ,                                     X
          CODEREG=R12
BRTESTLH CSECT
BRTESTLH AMODE 31
BRTESTLH RMODE ANY
*
BEGIN    DS    0H
*
* SEND MESSAGE INDICATING COMMENCEMENT
*
          EXEC CICS SEND                                     X
          FROM(BRTEXT1)                                     X
          LENGTH(BRTEXTL1)                                   X
          ERASE                                             X
          WAIT
*
*   PUT Container SESSIONDATA for user session data
*
          MVC    STRSDATA,CC1SDATA      Copy session data
          LA     R0,L'STRSDATA          Get session data length
          ST     R0,STRSDATL            Set the length
*
          EXEC   CICS PUT CONTAINER('SESSIONDATA') CHANNEL('ABC') X
          FROM(STRSDATA) LENGTH(STRSDATL)
          RESP(STRQRESP)
*
          MVC    STRERMSG,CC1MSG02      PUT container error message
          CLC    STRQRESP,DFHRESP(NORMAL) Command ok?
          BNE    CC1SENDM               No, send error message

```

```

*
*      PUT Container PARMCONTAINER for session parameters
*
      MVC   STRSPARM,CC1SPARM      Copy session parm
      LA    R0,L'CC1SPARM         Get session parm length
      ST    R0,STRSPRML           Set the length
*
      EXEC  CICS PUT CONTAINER('PARMCONTAINER') CHANNEL('ABC')      X
      FROM(STRSPARM) FLENGTH(STRSPRML)                             X
      RESP(STRQRESP)
*
      MVC   STRERMSG,CC1MSG02      PUT container error message
      CLC   STRQRESP,DFHRESP(NORMAL) Command ok?
      BNE   CC1SENDM              No, send error message
*
*      LINK to OSB CICS using Channel to pass session PARMS and
*      user session data.
*
      EXEC  CICS LINK PROGRAM('S6BCSSC2') CHANNEL('ABC')
*
*SEND MESSAGE INDICATING LINK RETURN
*
      EXEC CICS SEND                                X
      FROM(BRTEXT2)                                X
      LENGTH(BRTEXTL2)                             X
      ERASE                                         X
      WAIT
      B      BREND
*
*      Send message and return to CICS
*
CC1SENDM EXEC  CICS SEND FROM(STRERMSG) LENGTH(CC1ERMLN)
*
*      Return to CICS
*
BREND      EXEC  CICS RETURN
*
      LTORG ,                                     FORCE OUT ASSEMBLY LITERALS
*
CC1SPARM DC   CL16'U=USR10,P=PUSR10'
CC1SDATA DC   CL36'This is user session data'
CC1ERMLN DC   Y(L'STRERMSG)
*
BRTEXTL1 DC   Y(L'BRTEXT1) LENGTH OF LINK MESSAGE
BRTEXT1  DC   C'BRTESTLH - ABOUT TO LINK TO OSB/CICS'
BRTEXTL2 DC   Y(L'BRTEXT2) LENGTH OF RETURN MESSAGE
BRTEXT2  DC   C'BRTESTLH- RETURNED SUCCESSFULLY FROM OSB/CICS'
CC1MSG02 DC   CL20'PUT CONTAINER error'
*
      DFHREGS ,                                     EQUate registers
***>
*****>THE FOLLOWING DSECTS ALL EXIST IN CICS DYNAMIC STORAGE<*****
***>
      DFHEISTG ,
*
STRSDATL DS    F
STRSDATA DS    CL(L'CC1SDATA)

```

```

STRSPRML DS      F
STRSPARM DS      CL(L'CC1SPARM)
STRERMSG DS      CL20
          END     BRTESTLH

```

Using EXEC CICS XCTL to Start a Session

To start a TIBCO Object Service Broker session using **EXEC CICS XCTL**, establish a COMMAREA and transfer to the appropriate TIBCO Object Service Broker CICS client program.

The following program shows the use of **EXEC CICS XCTL**, which accesses the TIBCO Object Service Broker CICS client program **S6BCSSC2** and does not return to the assembler application.

```

BRTESTX  TITLE      'OSB-CICS XCTL EXAMPLE PROGRAM'
* * * * *
*                               BRTESTX                               *
* * * * *
* FUNCTION: THIS PROGRAM SERVES AS AN EXAMPLE OF HOW OSB-CICS CAN   *
*            BE ACCESSED FROM A USER APPLICATION PROGRAM.  THIS    *
*            EXAMPLE PERFORMS A TRANSFER OF CONTROL TO OSB-CICS AND *
*            PASSES PARAMETERS THROUGH A COMMAREA.  THE MESSAGES SENT *
*            FROM THIS PROGRAM INDICATE THE POINT AT WHICH THIS    *
*            PROGRAM IS PROCESSING.  FURTHER COMMENTS CAN BE FOUND  *
*            AT EACH SECTION.                                       *
* * * * *
* LINKAGE:-                                                         *
*   STANDARD CICS.                                                  *
* * * * *
*                               SPACE                               *
BRTESTX  DFHEIENT ,                                           X
          CODEREG=R12
BRTESTX  CSECT
BRTESTX  AMODE 31
BRTESTX  RMODE ANY
*
          B      BEGIN-BRTESTX(,R12)
BEGIN    DS      0H
*
* SEND MESSAGE INDICATING COMMENCEMENT
*
          EXEC   CICS SEND                                           X
                FROM(BRTEXT1)                                       X
                LENGTH(X'0024')                                     X
                ERASE                                              X
                WAIT
*
*MOVE LENGTH OF PARMS INTO COMMAREA
* # NOTE # THIS IS THE LENGTH OF THE PARM STRING ONLY.
*           THE COMMAREA LENGTH MUST INCLUDE AT MINIMUM, THE PARM

```

```

*           LENGTH PLUS THE LENGTH OF THE PARM LENGTH FIELD ITSELF.
*
*           MVC      HURNPRML,=X'001B'
*
*MOVE PARMS INTO COMMAREA. RULE NAME AND PARMS SPECIFIED HERE ARE
*JUST EXAMPLES.
*
*           MVC      HURNPARM,=C'RULE=TESTX(3245,BANK,DATES)'
*
*TRANSFER CONTROL TO OSB-CICS USING COMMAREA TO PASS RULES,PARMS
*
*           EXEC     CICS XCTL                                X
*                   PROGRAM('S6BCSSC2')                      X
*                   COMMAREA(HURNCOMA)                        X
*                   LENGTH(X'00A3')
*
* RETURN TO CICS
*
BREND      EXEC     CICS RETURN
           SPACE
           LTORG          FORCE OUT ASSEMBLY LITERALS
BRTEXTL1 DC     Y(L'BRTEXT1)          LENGTH OF LINK MESSAGE
BRTEXT1  DC     C'BRTESTX - ABOUT TO XCTL TO OSB-CICS'
           COPY     REGEQU
***>                                     <***
*****>THE FOLLOWING DSECTS ALL EXIST IN CICS DYNAMIC STORAGE<*****
***>                                     <***
*
* DSECT FOR BRTESTX
*
DFHEISTG DSECT
*
TERM      DS          CL4          TERMINAL ID
OSBCOMA DS          0F          COMMAREA FOR XCTL
OSBPRML DS          H          LENGTH OF PARMS IN HURNPARM
OSBPARM DS          CL161        PARMS (CAN BE LARGER)
OSBCOML EQU        *-OSBCOMA    LENGTH OF DSECT
           SPACE
           END          BRTESTX

```

Using EXEC CICS XCTL to Start a Session with Channel

To start a TIBCO Object Service Broker session with **EXEC CICS XCTL**, create a Channel with the desired Containers and transfer to the appropriate TIBCO Object Service Broker CICS client program.

The following program shows the use of **EXEC CICS XCTL**, which accesses the TIBCO Object Service Broker CICS client program **S6BCSSC2** and does not return to the assembler application.

```

BRTESTXH TITLE 'OSB-CICS XCTL EXAMPLE PROGRAM'
* * * * *
*                                     BRTESTXH                                     *
*
* FUNCTION: THIS PROGRAM SERVES AS AN EXAMPLE OF HOW OSB-CICS CAN
*           BE ACCESSED FROM A USER APPLICATION PROGRAM. THIS
*           EXAMPLE PERFORMS A TRANSFER OF CONTROL TO OSB-CICS AND
*           PASSES PARAMETERS TO OSB THROUGH A CHANNEL. THE
*           MESSAGES SENT FROM THIS PROGRAM INDICATE THE POINT AT
*           WHICH THIS PROGRAM IS PROCESSING. FURTHER COMMENTS CAN
*           BE FOUND AT EACH SECTION.
*
* LINKAGE:-
*   STANDARD CICS.
* * * * *
*           SPACE
BRTESTXH DFHEIENT ,                                     X
*           CODEREG=R12
BRTESTXH CSECT
BRTESTXH AMODE 31
BRTESTXH RMODE ANY
*
BEGIN      DS      0H
*
* SEND MESSAGE INDICATING COMMENCEMENT
*
*           EXEC CICS SEND                                     X
*           FROM(BRTEXT1)                                     X
*           LENGTH(BRTEXTL1)                                  X
*           ERASE                                             X
*           WAIT
*
* PUT Container SESSIONDATA for user session data
*
*           MVC      STRSDATA,CC1SDATA      Copy session data
*           LA       R0,L'STRSDATA         Get session data length
*           ST       R0,STRSDATL           Set the length
*
*           EXEC     CICS PUT CONTAINER('SESSIONDATA') CHANNEL('ABC')      X
*           FROM(STRSDATA) FLENGTH(STRSDATL)                                X
*           RESP(STRQRESP)
*
*           MVC      STRRMSG,CC1MSG02      PUT container error message
*           CLC      STRQRESP,DFHRESP(NORMAL) Command ok?
*           BNE      CC1SENDM              No, send error message
*
* PUT Container PARMCONTAINER for session parameters
*
*           MVC      STRSPARM,CC1SPARM      Copy session parm
*           LA       R0,L'CC1SPARM         Get session parm length
*           ST       R0,STRSPRML           Set the length
*
*           EXEC     CICS PUT CONTAINER('PARMCONTAINER') CHANNEL('ABC')      X
*           FROM(STRSPARM) FLENGTH(STRSPRML)                                X
*           RESP(STRQRESP)
*

```

```

        MVC   STRERMSG,CC1MSG02      PUT container error message
        CLC   STRQRESP,DFHRESP(NORMAL)  Command ok?
        BNE   CC1SENDM              No, send error message
*
*   Transfer control to OSB CICS using Channel to pass session PARMS
*   and user session data.
*
        EXEC  CICS XCTL PROGRAM('S6BCSSC2') CHANNEL('ABC')
*
*   Send message and return to CICS
*
CC1SENDM EXEC  CICS SEND FROM(STRERMSG) LENGTH(CC1ERMLN)
        EXEC  CICS RETURN
*
        LTORG ,                      FORCE OUT ASSEMBLY LITERALS
*
CC1SPARM DC    CL16'U=USR10,P=PUSR10'
CC1SDATA DC    CL36'This is user session data'
CC1ERMLN DC    Y(L'STRERMSG)
*
BRTEXTL1 DC    Y(L'BRTEXT1) LENGTH OF LINK MESSAGE
BRTEXT1  DC    C'BRTESTXH - ABOUT TO XCTL TO OSB-CICS'
CC1MSG02 DC    CL20'PUT CONTAINER error'
*
        DFHREGS ,                      EQUate registers
***>
*****>THE FOLLOWING DSECTS ALL EXIST IN CICS DYNAMIC STORAGE<*****
***>
        DFHEISTG ,
*
TERM      DS      CL4
STRSDATL  DS      F
STRSDATA  DS      CL(L'CC1SDATA)
STRSPRML  DS      F
STRSPARM  DS      CL(L'CC1SPARM)
STRERMSG  DS      CL20
END       BRTESTSH

```

Passing the COMMAREA Between a TIBCO Object Service Broker CICS Client and a Session

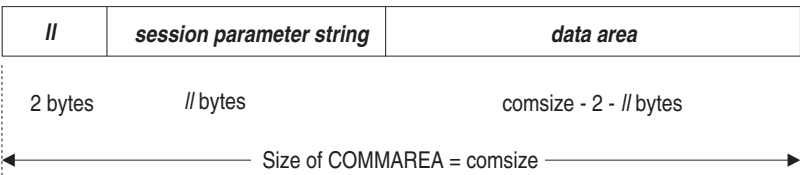
The CICS COMMAREA is used to:

- Pass data from the TIBCO Object Service Broker CICS client to the session at session startup
- Pass data to CICS routines called by rules
- Return data back to the TIBCO Object Service Broker CICS client at session end, when used by rules
- Return error messages generated by TIBCO Object Service Broker or by the [ENDMSG](#) tool

Whether the TIBCO Object Service Broker CICS client is non-seamless or seamless determines how the COMMAREA must be set up.

Non-Seamless COMMAREA

The non-seamless COMMAREA is arranged as shown in the following illustration:



The non-seamless TIBCO Object Service Broker client passes the session parameter string and the data area to the session when the session is started. Using this method, the client can pass data to a session through the arguments to the RULE parameter in the session parameter string or in the data area.

COMMAREA Requirements For the Non-Seamless TIBCO Object Service Broker Interface

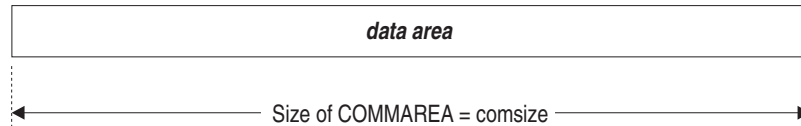
Before you use any of the CICS instructions to invoke a non-seamless TIBCO Object Service Broker client, the invoking CICS program must set up a COMMAREA that meets the following requirements:

- The non-seamless TIBCO Object Service Broker client supports a session parameter string and an optional data area, so the COMMAREA must be large enough to contain the parameters you pass to TIBCO Object Service Broker (preceded by a two-byte field containing the length of the parameters) and the data itself.

- The COMMAREA should be a minimum of 256 bytes long to accommodate error messages.
- The parameter length is a two-byte binary field (marked // in the previous figure), which must precede the session parameter string.
- The *session parameter string* must follow standard TIBCO Object Service Broker syntax, with brackets and commas where required. The *session parameter string* must not contain embedded nulls (0x00 characters), because a null is interpreted as the end of the string.
- The value in the parameter length field must be at least the total length of the parameters, including the parentheses and commas. The value does not include the length of the parameter length field itself.
- The data area must directly follow the session parameter string.

Seamless COMMAREA

The seamless TIBCO Object Service Broker client uses the entire externally established COMMAREA as a data area to be passed to the session when the session is started, as shown in the following illustration:



COMMAREA Requirements for the Seamless TIBCO Object Service Broker Interface

The COMMAREA should be a minimum of 256 bytes long to accommodate error messages.

Retrieving the COMMAREA in a Rule

Using TIBCO Object Service Broker rules, you can access only the data area portion of the COMMAREA. If the client is non-seamless, you can access only the data area following the session parameter string. If you use the seamless interface, you can access the data area consisting of the entire COMMAREA. The data area of the COMMAREA can be accessed using MAP tables or using the [\\$GETENVCOMMAREA](#) tool.

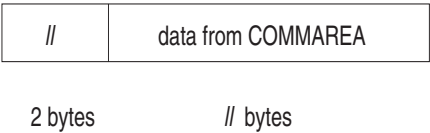
Using \$GETENVCOMMAREA

\$GETENVCOMMAREA returns the data portion of the COMMAREA as a syntax V string including the two byte // length header. The maximum length of the returned string is 31K. To avoid local variable storage overflow, consider increasing the values of the session parameters EXECLOCALSIZE and EXECSTACKSIZE. Settings of 128K for both variables are usually sufficient.

The syntax for using the \$GETENVCOMMAREA tool is:

```
COMM = $GETENVCOMMAREA(0);
```

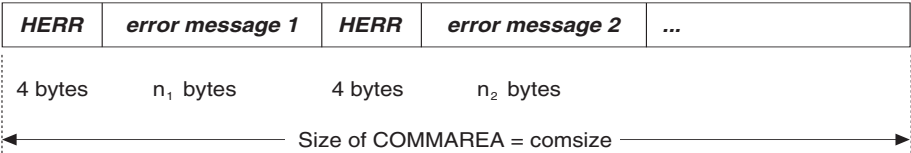
The value returned in COMM is as follows:



Error Messages in the COMMAREA

If the session terminates abnormally or if you use the ENDMSG tool to generate an error message, the message is put into the COMMAREA starting at the first byte, overlaying other data in that area. It is the responsibility of the CICS program that subsequently receives control to check for the four-byte error token HERR at the start of the COMMAREA.

The format of the error message is shown in the following illustration:



If the error messages cannot fit into the COMMAREA, it is truncated. To avoid truncating error messages, the size of the COMMAREA should be at least 256 bytes. For diagnostic purposes, a message that does not fit is sent to the console by WTO and appended to the CICS joblog.

See Also *TIBCO Object Service Broker Managing Data* about MAP tables.
 TIBCO Object Service Broker Shareable Tools about the tools.
 TIBCO Object Service Broker Parameters about parameters.

How Can Data Be Returned

Returning Data From TIBCO Object Service Broker to CICS

Data can be returned by rules to the COMMAREA by using:

- MAP tables to overlay and replace records in the COMMAREA
- The [\\$SETENVCOMMAREA](#) tool to assign a string to the entire COMMAREA

Steps to Returning an Occurrence

To return an occurrence to the CICS program from TIBCO Object Service Broker:

1. Prepare the COMMAREA in your CICS program.
2. Access TIBCO Object Service Broker using `EXEC CICS LINK, EXEC CICS START,` or `EXEC CICS XCTL.`

Refer to [Passing the COMMAREA Between a TIBCO Object Service Broker CICS Client and a Session on page 85](#).

3. Use TIBCO Object Service Broker rules to return data to the COMMAREA.

When you return to CICS, you can retrieve the data from the COMMAREA. If you have a CICS MRO (Multi-Region Option) system, your transaction must be defined to run on the same CICS MRO region as the TIBCO Object Service Broker startup module.

Using MAP Tables to Return Data

MAP tables are a convenient way to access the COMMAREA that is passed to the session and returned by it. Use the System Interpreted Table [@SESSION](#) to obtain and manipulate the pointer to the COMMAREA. For both seamless and non-seamless clients, the value of [@SESSION.COMMHANDLE](#) is the address of the COMMAREA, and [@SESSION.COMMLENGTH](#) is its length. Refer to [Non-Seamless COMMAREA on page 85](#) and [Seamless COMMAREA on page 86](#) for the layout of the COMMAREAs.

MAP tables support `FORALL`, `GET`, and `REPLACE` statements. All MAP tables are parameterized by a memory address, which you use to position your MAP table access to a location within the COMMAREA.

Using \$SETENVCOMMAREA to Return Data

The [\\$SETENVCOMMAREA](#) tool can be used to assign a string to the entire COMMAREA. The syntax is:

length=\$SETENVCOMMAREA(*value*, *segment#*);

where:

<i>length</i>	On return, contains the number of bytes available in the COMMAREA. If a COMMAREA does not exist, the length remaining is zero.
<i>value</i>	The data to be passed.
<i>segment#</i>	The number of the segment where the tool is to store the data. The value is always 0 for a CICS environment.

Example Usage of \$SETENVCOMMAREA

For example, the rules statement:

LEN = \$SETENVCOMMAREA(*STRINGOUT*, 0)

sets the entire COMMAREA to contain the string *STRINGOUT*, and sets *LEN* to the number of bytes left remaining in the COMMAREA.

See Also *TIBCO Object Service Broker for z/OS Installing and Operating* about setting up TIBCO Object Service Broker to use CICS MRO.

TIBCO Object Service Broker Managing Data about MAP tables.

TIBCO Object Service Broker Shareable Tools about the tools and System Interpreted Tables.

Performing CICS Functions at Session End

At session end, the default action taken by the TIBCO Object Service Broker CICS client program is to issue an **EXEC CICS RETURN**. You can execute a CICS command-level program written in COBOL, PL/I, assembler, or C at the end of your TIBCO Object Service Broker session either directly or through a CICS transid. Two methods are provided:

- Supply the **SESSIONENDACTION** and **SESSIONENDVALUE** session parameters to TIBCO Object Service Broker at session startup
- CALL the [\\$SETSESSIONEND](#) tool, which overrides any value set by the session parameters

Both methods require that you supply the **COMMAREA**: TIBCO Object Service Broker does not create a **COMMAREA** for this purpose. In either case, the CICS program or the CICS transid must be defined to CICS.

Starting a CICS Transaction

You can specify a CICS transaction to start after termination of the session and the TIBCO Object Service Broker CICS client. For non-seamless CICS clients, specify this using the following session parameters:

```
SESSIONENDACTION=START, SESSIONENDVALUE=transid
```

Alternatively, for all types of CICS clients, you can use the [\\$SETSESSIONEND](#) tool from within a rule:

```
CALL $SETSESSIONEND('START', 'transid');
```

The **COMMAREA** and **TERMINAL** are passed to the started CICS transaction.

Transferring to a CICS Program

You can specify a CICS program to be **XCTL**ed to after termination of the session and the TIBCO Object Service Broker CICS client. For the non-seamless CICS client, specify this using the following session parameters:

```
SESSIONENDACTION=XCTL, SESSIONENDVALUE=program_name
```

In all cases, [\\$SETSESSIONEND](#) can be used within a rule. The syntax is:

```
CALL $SETSESSIONEND('XCTL', 'program_name');
```

The **COMMAREA** is passed to *program_name*, which must be defined to CICS.

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

Calling External Routines

Under CICS, rules can use the CALL statement to call CICS programs that use CICS services or external routines that use OS linkage conventions. Refer to [Chapter 10, Accessing External Routines, on page 131](#) for detailed information about external routine usage.

Calling an External CICS Routine

You can access an external CICS routine using a CALL with the syntax:

```
CALL program_name;
```

where *program_name* is the CICS program defined in the PPT table and located in the DFHRPL library.

COMMAREA Preparation

If TIBCO Object Service Broker is invoked by a user application that defines a COMMAREA, it passes that COMMAREA to the CICS external routine. The COMMAREA can be modified using MAP tables or the [\\$SETENVCOMMAREA](#) tool before invoking the CICS external routine.

Requirements for Calling an External CICS Routine

- The routine must be defined in the ROUTINES table and the value in the **LOADNAME** field must be the same as the **NAME** field.
- The language specified in the **LANGUAGE** field of the ROUTINES table must be CICS—not the language in which the routine is written, except in the case of routines with OS linkage. Refer to [Calling An External Routine With OS Linkage on page 92](#).
- Error handling must return the routine to TIBCO Object Service Broker.

Restrictions for Calling an External CICS Routine

- If the TIBCO Object Service Broker session is initiated without a COMMAREA, the external routine *cannot* create one and pass it to TIBCO Object Service Broker for use.
- You cannot specify arguments in the ARGUMENTS table; therefore, the external routines cannot be defined as functions.

- No parameter is allowed for the CALL. Data must be passed in the COMMAREA.
- The CICS program must not contain TIBCO Object Service Broker access statements. Refer to [Chapter 21, Coding TIBCO Object Service Broker Access Statements, on page 339](#) for more information.
- Stacked sessions are not allowed. A new session cannot be started through the use of an external routine that invokes a TIBCO Object Service Broker CICS client program from within an existing TIBCO Object Service Broker CICS client session.
- The **EXEC CICS HANDLE ABEND** command is not allowed. All other **EXEC CICS HANDLE error_condition** commands are permitted.
- When issuing an **EXEC CICS ABEND ABCODE** command to terminate the CICS program abnormally, do not start your four-character code with an “H”. These codes are reserved for internal TIBCO Object Service Broker use.
- When issuing an **EXEC CICS ABEND ABCODE** command from a CICS external routine, the option CANCEL must not be specified. Use of this option may cause your session to hang as TIBCO Object Service Broker’s error handling is bypassed.

Calling An External Routine With OS Linkage

Rules can call external routines defined using the ROUTINES and ARGUMENTS tables. External routines with OS linkage conventions can also be defined as functions. Refer to [Chapter 10, Accessing External Routines, on page 131](#) for more information.

Usage Note

For routines with OS linkage, the language specified in the **LANGUAGE** field of the ROUTINES table must be the language in which the routine is written—do *not* specify CICS.

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

CICS Channels and Containers in the TIBCO Object Service Broker CICS Session Environment

You can do the following using the CICS Channel:

- Pass data to the TIBCO Object Service Broker CICS session at session startup.
- Pass data to CICS routines that are called by rules.
- Return data from the TIBCO Object Service Broker CICS session at session end.
- Return the error messages generated by TIBCO Object Service Broker or by the ENDMSG tool.

CICS Channel and Container Tools

TIBCO Service Gateway for CICS provides a set of tools for handling CICS Channels and containers. See the following table for the tools along with the equivalent CICS API commands.

CICS API Command	Object Service Broker Tool
EXEC CICS ASSIGN CHANNEL(<i>data-area</i>)	\$SHOWCHANNEL(<i>channel_name</i>)
EXEC CICS STARTBROWSE CONTAINER CHANNEL(<i>name</i>) BROWSETOKEN(<i>data-value</i>)	\$BRCONTAINER(<i>channel, container_list</i>)
EXEC CICS GETNEXT CONTAINER(<i>name</i>) BROWSETOKEN(<i>data-value</i>)	The previous tool includes this function.
EXEC CICS ENDBROWSE CONTAINER BROWSETOKEN(<i>data-value</i>)	The previous tool includes this function.
EXEC CICS GET CONTAINER(<i>name</i>) CHANNEL(<i>name</i>) INTO(<i>data-area</i>) FLENGTH(<i>data-area</i>)	\$GETCONTAINER(<i>channel, container, area, length</i>)
EXEC CICS PUT CONTAINER(<i>name</i>) CHANNEL(<i>name</i>) FROM(<i>data-area</i>) FLENGTH(<i>data-area</i>)	\$PUTCONTAINER(<i>channel, container, area, length</i>)
EXEC CICS MOVE CONTAINER(<i>name</i>) AS(<i>name</i>) CHANNEL(<i>name</i>) TOCHANNEL(<i>name</i>)	\$MOVECONTAINER(<i>fromchannel, fromcontainer, tochannel, tocontainer</i>)

CICS API Command	Object Service Broker Tool
EXEC CICS DELETE CONTAINER(<i>name</i>) CHANNEL(<i>name</i>)	\$DELCONTAINER(<i>channel</i> , <i>container</i>)
	\$SETCHANNEL(<i>channel_name</i>)
EXEC CICS LINK PROGRAM(<i>name</i>) CHANNEL(<i>name</i>)	
EXEC CICS RETURN TRANSID(<i>name</i>) CHANNEL(<i>name</i>)	
EXEC CICS START TRANSID(<i>name</i>) CHANNEL(<i>name</i>)	
EXEC CICS XCTL PROGRAM(<i>name</i>) CHANNEL(<i>name</i>)	

For a detailed description of the tools, refer to the *TIBCO Object Service Broker Shareable Tools* manual. For general information and the rules that govern the scope of channels, see the *CICS Transaction Server for z/OS CICS Application Programming Guide*.

The following table describes the tasks performed by the tools.

Tool	Task
\$SHOWCHANNEL	Returns the 16-character name of the session’s current channel if one exists. Otherwise, returns blanks.
\$BRCONTAINER	Lists the 16-character container names and displays the count of the containers associated with the channel. This tool combines the functions of the following CICS API commands: <ul style="list-style-type: none">• STARTBROWSE CONTAINER• GETNEXT CONTAINER• ENDBROWSE CONTAINER
\$GETCONTAINER	Retrieves the data associated with the specified channel container.
\$PUTCONTAINER	Places data in a container associated with the specified channel.
\$MOVECONTAINER	Moves a container and its contents from one channel to another. Afterwards, the source container no longer exists.

Tool	Task
\$DELCONTAINER	Deletes a container from a channel and discards the container's data, if any.
\$SETPCHANNEL	Nominates a channel for passing to a CICS routine when called in a rule or for passing to a program or transaction through the SESSIONEND action. A blank channel name cancels any previously nominated channel.

Channel Scope

In the TIBCO Object Service Broker CICS execution environment are two session modes, as follows:

- **Conversational mode (CICSPSEUDOCONVERSE=N)** — In this mode, the entire session is considered one program because no EXEC CICS RETURN command is issued until at the end of the session. Therefore, the Current Channel, if one exists, and the Channels created during the session are available throughout the session.

The sessions that are started through LINK or XCTL from a CICS program run in conversational mode.

- **Pseudo-conversational mode (CICSPSEUDOCONVERSE=Y)** — In this mode, a screen display results in EXEC CICS RETURN with TRANSID and a CHANNEL option. Therefore, the Current Channel, if one exists, is passed on to the next leg of the pseudo-conversation. However, the Channels created during the session are not passed and are hence out of scope.

You can configure both modes with the Execution Environment parameter CICSPSEUDOCONVERSE=Y | N.

Predefined Container Names

Two Execution Environment parameters name the container for passing the session parameter to a TIBCO Object Service Broker session and for returning the error messages that are generated by the session, as follows:

- **CICSPCONTAINER** — This container passes the session parameter to an Object Service Broker session for startup. The default name is PARMCONTAINER.
- **CICSECONTAINER** — This container enables the Object Service Broker session to return error messages to the invoker of the session. The default name is HERRCONTAINER.

For details, see the *TIBCO Object Service Broker Parameters* manual. You can pass user data to and from the session in separate containers with names of your choice.

Chapter 8

Using the TIBCO Service Gateway for IMS TM

This chapter describes how to run and set TIBCO Object Service Broker sessions under IMS TM, using the Service Gateway for IMS TM.

Topics

- [How to Run IMS TM Applications, page 98](#)
- [Selecting a TIBCO Object Service Broker IMS TM Program Style, page 101](#)
- [Starting a TIBCO Object Service Broker Session, page 103](#)
- [Terminal Changes at Session Startup, page 105](#)
- [Passing Data to TIBCO Object Service Broker IMS TM Sessions, page 106](#)
- [Input Message Segment Overview, page 108](#)
- [Returning Data from TIBCO Object Service Broker to IMS TM, page 112](#)
- [Passing Control to an IMS Transaction at Session End, page 114](#)
- [Ensuring Message Queue/Database Consistency, page 117](#)
- [Customizing TIBCO Object Service Broker IMS TM Client Programs, page 119](#)
- [Getting Access to IMS TM Data, page 120](#)

How to Run IMS TM Applications

Functional Overview

You can run IMS TM applications using the Service Gateway for IMS TM which enables you to use IMS TM clients to run IMS TM applications that access TIBCO Object Service Broker from an IMS TM Message Processing Region (MPR). You can start the clients by either conversational or non-conversational IMS TM transactions.



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

Using TIBCO Object Service Broker IMS TM Client Programs

When an IMS TM MPR has established communication with a Native Execution Environment, you can establish a TIBCO Object Service Broker session by scheduling an IMS TM transaction associated with one of the TIBCO Object Service Broker IMS TM client programs. There are eight client programs with the format S6BIM_{xxx}. These are described in [Selecting a TIBCO Object Service Broker IMS TM Program Style on page 101](#).

Each S6BIM_{xxx} program takes its messages from the IMS Message Queue and sends these, together with session parameters, to the Native Execution Environment, where a session is started. The session runs the application as determined by the startup rule and, upon completion, returns output messages and session and client directives to the TIBCO Object Service Broker IMS TM client S6BIM_{xxx}. The client terminates and the MPR schedules another transaction.

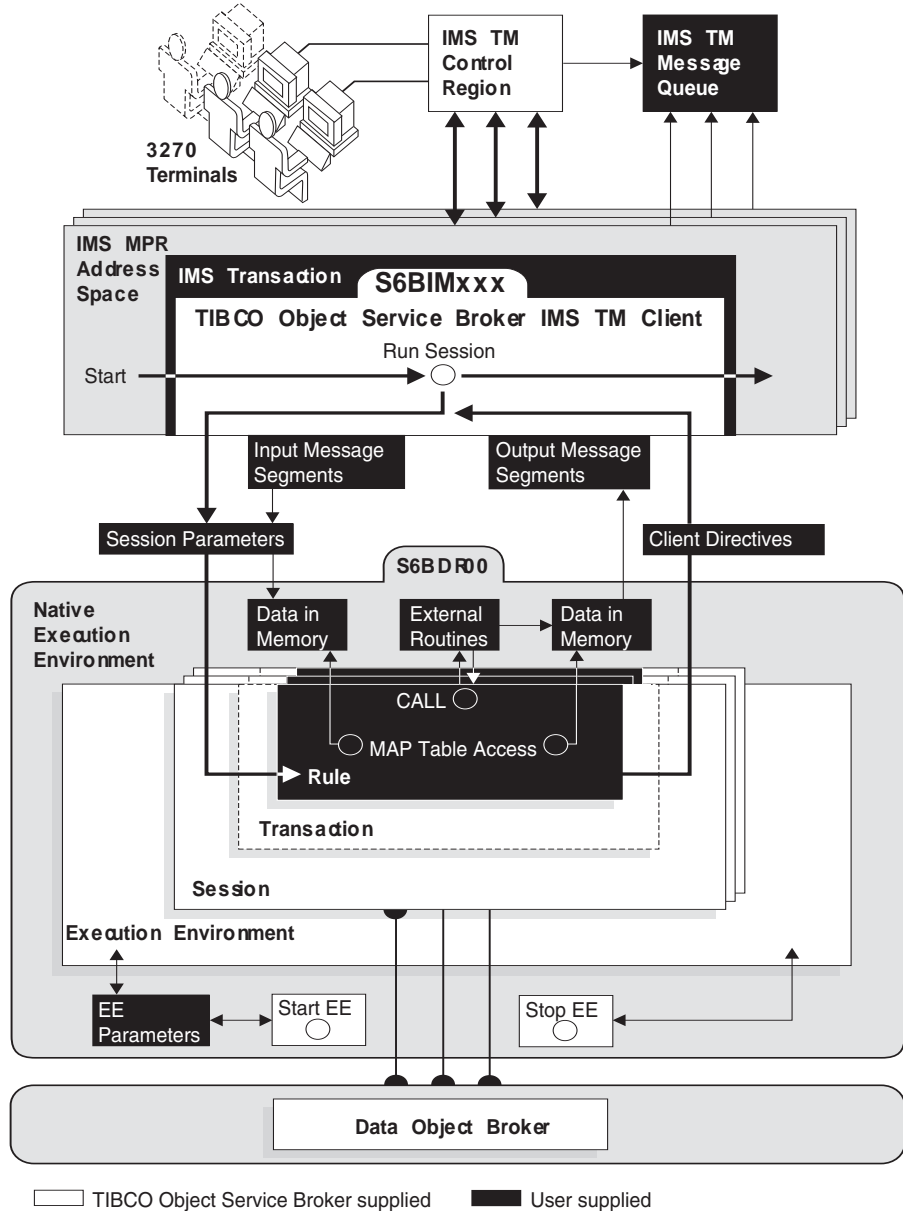


The TIBCO Object Service Broker Call Level Interface does not support an IMS TM environment.

How a Client Session is Established

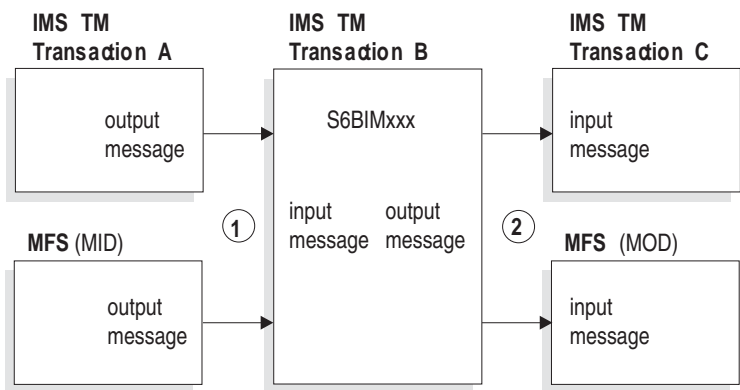
When a client session is being established, the IMS TM client program in the MPR first locates the Native Execution Environment and passes it the session start request. Since IMS TM is a multiple-session TP monitor, the Native Execution Environment is established separately, typically under operational control.

The following illustration shows how a typical supplied IMS TM client establishes a session from the MPR in a Native Execution Environment:



IMS TM and TIBCO Object Service Broker Interaction

A user IMS transaction can start a TIBCO Object Service Broker based IMS transaction and its associated IMS TM client by sending an output message to the TIBCO Object Service Broker based IMS transaction, as shown below:



Similarly, a terminating TIBCO Object Service Broker session can pass data to a subsequent IMS transaction or Message Formatting Services (MFS) MODname to be scheduled in the IMS environment. Facilities are provided in the rules language to specify the next IMS transaction or MFS MODname and the output message segments.

Replacing IMS TM Programs with TIBCO Object Service Broker Rules

Existing IMS TM programs can be replaced by TIBCO Object Service Broker rules without changing interfaces to the previous and succeeding programs.



Using this example, transaction A invokes transaction B and transaction B invokes transaction C. Initially, these are all IMS transactions. Over time, these transactions can be replaced by TIBCO Object Service Broker rules that implement the same, or enhanced functionality. For example, in the illustration above, the program in transaction B is replaced with a TIBCO Object Service Broker IMS TM client.

See Also *TIBCO Object Service Broker for z/OS Installing and Operating* about installing the IMS TM component of TIBCO Object Service Broker.
TIBCO Object Service Broker Programming in Rules about writing rules.

Selecting a TIBCO Object Service Broker IMS TM Program Style

You can choose from among eight different client styles. Each style corresponds to a different IMS TM program name. These programs are installed as part of the TIBCO Object Service Broker IMS TM interface.

TIBCO Object Service Broker IMS TM Client Programs

The following table lists the available client programs. The IMS Program Type column shows if the client is conversational (C) or non-conversational (NC).

Transaction Name	Client Program Name	TIBCO Object Service Broker user ID set by	IMS Program Type	Seamless
User supplied	S6BIMSC1	External user ID ^a	C	Y
S6BLOGON or user supplied	S6BIMSC2	External user ID ^a or USERID parameter	C	N
User supplied	S6BIMSN1	External user ID ^a	NC	Y
User supplied	S6BIMSN2	External user ID ^a or USERID parameter	NC	N
User supplied	S6BIMTC1	IMS TM transaction name	C	Y
User supplied	S6BIMTC2	IMS TM transaction name or USERID parameter	C	N
User supplied	S6BIMTN1	IMS TM transaction name	NC	Y
User supplied	S6BIMTN2	IMS TM transaction name or USERID parameter	NC	N
S6BDCKRN	S6BDCKRN	NA - Resume Trans	n/a	n/a

a. The external user ID is obtained from the terminal I/O PCB or can be set by your installation TIBCO Object Service Broker IMS TM exit routine (refer to [Customizing TIBCO Object Service Broker IMS TM Client Programs on page 119](#) for more information).

Choosing the Right TIBCO Object Service Broker IMS TM Client Program

Match your style requirements with the appropriate TIBCO Object Service Broker IMS TM client program name listed in the following table.

Client Program	Description
S6BIMSxx	User ID default is the external user ID.
S6BIMTxx	User ID default is the transaction name (refer to Setting Up the User Profile for Seamless Clients on page 18).
S6BIMxCx	IMS conversational transaction client: has a Scratch Pad Area (SPA) and supports TIBCO Object Service Broker screen I/O.
S6BIMxNx	IMS non-conversational transaction client: no SPA. TIBCO Object Service Broker supports TIBCO Object Service Broker screen I/O but you must install the IMS Physical Input Edit/Exit routine.
S6BIMxx1	Seamless interface, one data segment, uses IMS transaction name to determine startup rule.
S6BIMxx2	Non-seamless interface, first segment is session parameter string, second segment is user data. Use R= in session parameter string or user ID's user profile to set the first rule.



Seamless clients do not support the RULE parameter for session startup; the name of the rule is obtained from the user profile associated with the IMS transaction name. This rule is run with no arguments and with ACTION, SEARCH, BROWSE/NOBROWSE, and TEST/NOTEST parameters set based on user profile values or on the TIBCO Object Service Broker defaults. The rule obtains the input message segments using MAP tables or the [\\$GETENVCOMMAREA](#) tool.

See Also *TIBCO Object Service Broker Managing Data* about MAP tables.
TIBCO Object Service Broker Shareable Tools about the tools.
TIBCO Object Service Broker Parameters about parameters.

Starting a TIBCO Object Service Broker Session

How to Set Session Parameters

The TIBCO Object Service Broker IMS TM session parameter values are those of the Native Execution Environment connected to the MPR. These are determined according to the table shown in [Where to Specify Session Parameters on page 54](#) and are described in [Chapter 6, TIBCO Object Service Broker Sessions Under the Native Execution Environment, on page 51](#).

Using the IMS TM Terminal to Start a Session

To start a session from an IMS TM terminal, enter an IMS TM trancode associated with a TIBCO Object Service Broker IMS TM client. Non-seamless client programs use the screen input following the trancode as the *session parameter string*. No data is passed to the TIBCO Object Service Broker session.

Usage of the Supplied Trancode

The supplied trancode is **S6BLOGON**, which is associated with the program S6BIMSC2. S6BIMSC2 is non-seamless, permits screen I/O, and uses the external security user ID (which can be overridden with the USERID parameter). When you use parameters with the **S6BLOGON** transaction, the syntax is:

S6BLOGON *parameter*[=*value*][,]

where:

<i>parameter</i>	The parameter name or abbreviation (valid abbreviations are listed in <i>TIBCO Object Service Broker Parameters</i>). Use commas to separate multiple parameters.
<i>value</i>	The parameter value (if required) follows an equal sign (=).

Example

An example of the **S6BLOGON** transaction is:

S6BLOGHON U=USR09 , PASSWORD=USR09

This establishes a session with parameters USERID (abbreviated as U) and PASSWORD. You do not have to specify a user ID and password if you log in using **/SIGNON** (or an equivalent transaction). Since no rule is specified, the TIBCO Object Service Broker workbench appears, unless the user profile specifies a rule.

You can associate an IMS transaction identifier with a TIBCO Object Service Broker IMS TM client program and invoke it from the IMS TM terminal. Anything following the transaction name is placed in the first message segment. If the client program is conversational, the transaction name is in the SPA; if non-conversational, the transaction name is included in the first message segment.

Using Message Formatting Services (MFS) to Start a Session

To start a TIBCO Object Service Broker IMS TM transaction, MFS can schedule the transaction as a result of:

- Calling up a format screen using **/FORMAT MODName**
The Message Output Descriptor (MOD) specifies a user-defined transaction name associated with one of the TIBCO Object Service Broker IMS TM client programs.
- Standard MFS processing of a Message Input Descriptor (MID) originated by a non-conversational IMS program
- Deferred Message Switching originated by a conversational IMS program

Program-to-Program Message Switching to Start a Session

To start a TIBCO Object Service Broker IMS TM transaction by another application program, use an alternate PCB to perform a program-to-program message switch:

- Program-to-program message switching is supported from non-conversational application programs to non-conversational TIBCO Object Service Broker IMS TM client programs (**S6BIMxNx**).
- Deferred program switching is supported from conversational application programs to conversational TIBCO Object Service Broker IMS TM client programs (**S6BIMxCx**).

Terminal Changes at Session Startup

Extended Terminal Support

Users of IMS TM terminals can take advantage of different model terminal screen sizes and terminal extended attribute support. You can define the screen size in the IMS sysgen `TERMINAL` macro and re-link the IMS nucleus with `S6BDCSGN`, a new signon exit that includes the module `HDRSBDRC`. When you sign on to IMS and TIBCO Object Service Broker, the workbench appears with the proper size.

Additional Terminal Capabilities

Additional terminal capabilities with regard to extended attribute or extended data stream are handled by the `IMSSCREENATTRIBU` Execution Environment parameter. You can obtain and explicitly set the attributes using the `$GETOPT` and `$SETOPT` tools.

PF Key Changes

As of session startup, all TIBCO Object Service Broker tools when used under IMS TM make use of the following PF key assignments:

- PF24 replaces PF12 (for CANCEL)
- CLEAR replaces PF24 for (for RESHOW)

The changes are automatic and no user action is required. Also, IMS TM usually uses PF12 for print screen.

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

TIBCO Object Service Broker Parameters about the Execution Environment parameters.

Passing Data to TIBCO Object Service Broker IMS TM Sessions

Depending on the seamless/non-seamless and conversational/non-conversational style of the client, at session start your rules can read or write up to three message segments. The rules can use MAP tables or the [\\$GETENVCOMMAREA](#) and [\\$SETENVCOMMAREA](#) tools. The message segments are the SPA, the session parameter string segment, and the data segment. Every client sends the data segment to the session.

Using MAP Tables to Access Data

MAP tables are a convenient means of accessing the message segments that are passed to the session and returned by it. Use the System Interpreted Table [@SESSION](#) to obtain and manipulate the pointers to the message segments.

You can access the three input and output message segments by using the memory pointers:

- [@SESSION.SEG \$n\$ INHANDLE](#)
- [@SESSION.SEG \$n\$ OUTHANDLE](#)

where n is the segment number 0, 1, or 2. Segment 0 corresponds to the SPA, and is valid only for conversational style clients. The lengths of these message segments are in [@SESSION.SEG \$n\$ INLENGTH](#) and [@SESSION.SEG \$n\$ OUTLENGTH](#).

MAP tables support FORALL, GET, and REPLACE statements. All MAP tables are parameterized by a memory address that you use to position your MAP table access to any location within the message segment.

Using the \$GETENVCOMMAREA Tool to Access Data

[\\$GETENVCOMMAREA](#) returns all the message segment as a syntax V string, including the *llzz* or *llzzzz* components. The maximum length of the returned string is 31K. To avoid local variable storage overflow, consider increasing the values of the session parameters EXECLOCALSIZE and EXECSTACKSIZE. Setting each of these to 128K is usually more than sufficient.

The syntax is:

```
value = $GETENVCOMMAREA(segment#);
```

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

TIBCO Object Service Broker Shareable Tools about the tools and System Interpreted Tables.

TIBCO Object Service Broker Parameters about parameters.

Input Message Segment Overview

Message Segment Types

The TIBCO Object Service Broker IMS TM client style determines the number and format of message segments sent to the session. Only conversational clients use a Scratch Pad Area (SPA) designated as segment 0. Only non-seamless clients use a session parameter string designated as segment 1. All clients have a data segment.

Scratch Pad Area (SPA)

IMS TM uses the first eight bytes of the SPA for the trancode. Your application can use the remainder of the SPA (up to the maximum available SPA size) to pass data to a conversational IMS TM client program.

`$GETENVCOMMAREA(0)` returns the SPA (including the trancode) if it exists.

Session Parameter String Segment

The session parameter string is the first message segment for non-seamless (**S6BIMxx2**) TIBCO Object Service Broker IMS TM client programs. You can use this segment to pass data by specifying values to the rule in the RULE session parameter. `$GETENVCOMMAREA(1)` returns this segment for non-seamless IMS TM clients (if non-conversational, the session parameter string follows the trancode in the first eight bytes).

Data Segment

For seamless clients, data supplied by MFS or through program-to-program switching is in segment 1. `$GETENVCOMMAREA(1)` returns the data segment. If non-conversational, the data segment includes the trancode in the first eight bytes.

For non-seamless clients, data supplied by MFS or through program-to-program switching is in segment 2. `$GETENVCOMMAREA(2)` returns the data segment.

S6BIMxC1 Client Program Input Message Format

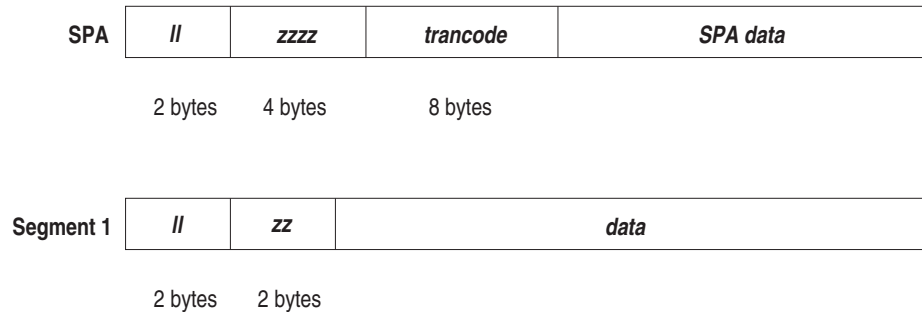
The S6BIMxC1 client programs are seamless and conversational as shown in the following table:

Interface Program	Conversational	Seamless	SPA ^a	Segment 1 ^b
S6BIMSC1	Y	Y	Y	User data
S6BIMTC1				

a. The SPA is segment 0, in the cases where it exists.

b. Segment 1 can contain either user data (seamless interface) or the session parameter string (non-seamless interface).

Input Message Format for S6BIMSC1 and S6BIMTC1



S6BIMxC2 Client Program Input Message Format

The S6BIMxC2 client programs are non-seamless and conversational, as shown in the following table:

Interface Program	Conversational	Seamless	SPA ^a	Segment 1 ^b	Segment 2 ^c
S6BIMSC2	Y	N	Y	Session	Data
S6BIMTC2				parameter string	

a. The SPA is segment 0, in the cases where it exists.

b. Segment 1 can contain either user data (seamless interface) or the session parameter string (non-seamless interface).

c. Segment 2 can contain only user data.

Input Message Format for S6BIMSC2 and S6BIMTC2

SPA	<div>//</div>	<div>zzzz</div>	<div>trancode</div>	<div>SPA data</div>
	2 bytes	4 bytes	8 bytes	
Segment 1	<div>//</div>	<div>zz</div>	<div>session parameter string</div>	
	2 bytes	2 bytes		
Segment 2	<div>//</div>	<div>zz</div>	<div>data</div>	
	2 bytes	2 bytes		

S6BIMxN1 Client Program Input Message Format

The S6BIMxN1 client programs are seamless and non-conversational, as shown in the following table:

Interface Program	Conversational	Seamless	SPA ^a	Segment 1 ^b
S6BIMSN1	N	Y	N	Data
S6BIMTN1				

- a. The SPA is segment 0, in the cases where it exists.
- b. Segment 1 can contain either user data (seamless interface) or the session parameter string (non-seamless interface).

Input Message Format for the S6BIMSN1 and S6BIMTN1

Segment 1	<div>//</div>	<div>zz</div>	<div>trancode</div>	<div>data</div>
	2 bytes	2 bytes	8 bytes	

S6BIMxN2 Client Program Input Message Format

The S6BIMxN2 client programs are non-seamless and non-conversational, as shown in the following table:

Interface Program	Conver-sational	Seamless	SPA ^a	Segment 1 ^b	Segment 2 ^c
S6BIMSN2	N	N	N	Session parameter string	Data
S6BIMTN2					

- a. The SPA is segment 0, in the cases where it exists.
- b. Segment 1 can contain either user data (seamless interface) or the session parameter string (non-seamless interface).
- c. Segment 2 can contain only user data.

Input Message Format for S6BIMSN2 and S6BIMTN2

Segment 1	//	zz	trancode	session parameter string
	2 bytes	2 bytes	8 bytes	
Segment 2	//	zz	data	
	2 bytes	2 bytes		

See Also *TIBCO Object Service Broker Shareable Tools* about the tools.
 TIBCO Object Service Broker Parameters about parameters.

Returning Data from TIBCO Object Service Broker to IMS TM

You can set the contents of the SPA (for conversational programs) and up to two output message segments in the I/O-PCB from TIBCO Object Service Broker rules using either MAP tables or the `$SETENVCOMMAREA` tool.

Example using `$SETENVCOMMAREA`

Although MAP tables are likely to be more useful for real applications, for brevity `$SETENVCOMMAREA` is used in the following example. Its syntax is:

`LEN = $SETENVCOMMAREA(message, segment)`

where:

<i>LEN</i>	The amount of space available in this segment after insertion
<i>message</i>	A string comprising the entire output message segment
<i>segment</i>	The segment number If <i>segment</i> =0, the string is inserted into the SPA. When the TIBCO Object Service Broker IMS TM client program inserts these messages into the I/O-PCB on termination, each message must have the format shown below, which includes at least a valid <i>llzz</i> field:

Segment 0 (SPA)	<i>ll</i>	<i>zzzz</i>	<i>trancode</i>	<i>value</i>
	2 bytes	4 bytes	8 bytes	
Segment 1 or 2	<i>ll</i>	<i>zz</i>	<i>value</i>	
	2 bytes	2 bytes		

At Session End

Depending on the actions that you require at session end, you must construct valid IMS output message segments. You must establish the length *ll* and preload the IMS control area *zz* or *zzzz* components before issuing `$SETENVCOMMAREA`. An example for segment 1 is:

```
LL_OUT = GENBIN(LENGTH(DATA) + 4, 2);
ZZ_OUT = GENBIN('00', 2);
STRING_OUT = LL_OUT || ZZ_OUT || DATA;
LEN = $SETENVCOMMAREA(STRING_OUT, SEGMENT);
```

where *LEN*, *LL_OUT*, *DATA*, *STRING_OUT*, *SEGMENT* are defined as local variables, with *SEGMENT* set to the segment number.

The trancode to which a message is to be sent is reserved in the first eight bytes of segment 1 if the destination is non-conversational. The first eight bytes are set by the **SWITCH** facility described in [Passing Control to an IMS Transaction at Session End on page 114](#).

The returned data is directed either to your terminal or to another IMS program as described in the following section.



Do *not* change the first 14 bytes of segment 0 (the SPA). Use MAP tables or the `GENBIN` tool to set up the first four bytes of segments 1 or 2.

See Also *TIBCO Object Service Broker Managing Data* about using MAP tables.
 TIBCO Object Service Broker Shareable Tools about the tools.
 TIBCO Object Service Broker Parameters about parameters.

Passing Control to an IMS Transaction at Session End

What are the Allowable Options for Passing Control

At session end, the TIBCO Object Service Broker IMS TM client program returns control of the terminal to the user by default. You can direct the message segments returned by the TIBCO Object Service Broker session directly to another IMS application program, or to a terminal via MFS. Specifically, you can perform:

- MFS output from a non-conversational TIBCO Object Service Broker IMS TM client program
- A program-to-program switch from a non-conversational IMS TM client program to another IMS non-conversational program
- A deferred message switch from a conversational TIBCO Object Service Broker IMS TM client program to another IMS conversational program
- An immediate program-to-program message switch from a conversational TIBCO Object Service Broker IMS TM client program to another IMS conversational program

What to Use to Direct the Destination of Message Segments

You can direct the destination of the message segments at session startup, or from rules, by:

- Supplying the `SESSIONENDACTION` and `SESSIONENDVALUE` session parameters to TIBCO Object Service Broker at session startup
- Using the `$SETSESSIONEND` tool within a rule

Non-Conversational MFS Output

All non-conversational TIBCO Object Service Broker IMS TM client programs (`S6BIMSN1`, `S6BIMSN2`, `S6BIMTN1`, and `S6BIMTN2`) can send data to the terminal at session end using MFS. The Message Output Descriptor (MOD) to be used can be specified using:

- Session parameters:
`SESSIONENDACTION=FORMAT`, `SESSIONENDVALUE=MODname`
- A rule:
`CALL $SETSESSIONEND('FORMAT', 'MODname');`

The `MODname` set by the rule replaces any value set by session parameters.

Non-Conversational Program-to-Program Switch

All non-conversational TIBCO Object Service Broker IMS TM client programs of the form **S6BIMxNx** can perform a program-to-program switch at session end. The IMS application transaction that is being switched to can be specified using:

- Session parameters:
SESSIONENDACTION=SWITCH, SESSIONENDVALUE=trancode
- A rule:
CALL \$SETSESSIONEND('SWITCH','trancode');

The trancode set by the rule replaces any value set by the session parameters. The trancode replaces the first eight bytes of the first message segment at session end.

Conversational Deferred Message Switch

All conversational TIBCO Object Service Broker IMS TM client programs (**S6BIMSC1**, **S6BIMSC2**, **S6BIMTC1**, and **S6BIMTC2**) can send data to the terminal at session end using MFS and switch to another IMS application transaction after the user responds to the terminal. You can specify this using:

- Session parameters. You can specify either the trancode or the MODname but not both:
SESSIONENDACTION=SWITCH, SESSIONENDVALUE=trancode
SESSIONENDACTION=FORMAT, SESSIONENDVALUE=MODname
- A rule. You can specify both the trancode and the MODname:
CALL \$SETSESSIONEND('SWITCH','trancode');
CALL \$SETSESSIONEND('FORMAT','MODname');

The value specified by [\\$SETSESSIONEND](#) supersedes any value set by the session parameters.

You must specify both the trancode and the MODname for a deferred message switch to occur at session end. The trancode specified replaces the first eight bytes of the SPA at session end.

Conversational Immediate Message Switch

All conversational TIBCO Object Service Broker IMS TM client programs (**S6BIMx**) can directly switch to another conversational IMS application transaction. You can specify the trancode of the IMS application transaction using:

- Session parameters:
\$SESSIONENDACTION=SWITCH, SESSIONENDVALUE=trancode
- A rule:
CALL \$SETSESSIONEND('SWITCH','trancode');

The trancode specified by the rule supersedes any value set by the session parameters. The trancode replaces the first eight bytes of the SPA at session end.

See Also *TIBCO Object Service Broker Programming in Rules* about writing rules.

TIBCO Object Service Broker Shareable Tools about the tools.

TIBCO Object Service Broker Parameters about parameters.

Ensuring Message Queue/Database Consistency

TIBCO Object Service Broker Supplied Facility

IMS TM includes message synchronization for messages destined to other transactions or to terminals as part of the commit processing for a transaction. To make this consistency available for TIBCO Object Service Broker transactions, the TIBCO Object Service Broker-supplied table @IMSDCTRXS helps users to keep track of the status of the most recently executed transaction.

@IMSDCTRXS Table

The @IMSDCTRXS table contains the following fields:

- **LTERM**
- **REGION**
- **CODE**
- **DATE**
- **TIME**
- **COMPLETED**

This table has a composite primary key of the **LTERM** and **REGION** fields. At the beginning of each session you can have a startup rule access this table to determine if any messages have to be re-sent, as indicated by the value in the fields of the table.

Retrieval of Information

Use the [\\$GETOPT](#) tool to retrieve the **LTERM**, **REGION**, **CODE**, and any other information required to make decisions about re-sending messages.

Sample Rules for Processing

A set of sample rules to do this processing is provided as part of the TIBCO Object Service Broker IMS TM interface. These rules are in the @SAMPLES library (Unit=IMSDC). The entry rule is SAMPLEIMSDCTR.

The comparison is done on IMS LTERM (logical terminal) names and time/date stamps. The message to be re-sent is stored in the table @IMSDCTRROUT in multiple segments of 1024 bytes.

@IMSDCTRXOUT is parameterized by LTERM, REGION, and SEGMENT:

- LTERM and REGION are the values specified in table @IMSDCTRXS. (REGION in this case refers to the name of the IMS Control Region from which the message is sent.)
- SEGMENT is numbered from 1 to the number of segments in the message.

If the startup rule is invoked and there is a message to be sent, the sample rules show how to send it back to IMS for processing by the TIBCO Object Service Broker program running in the MPR.

See Also *TIBCO Object Service Broker Programming in Rules* about writing rules and rules libraries.

TIBCO Object Service Broker Shareable Tools about the tools.

Customizing TIBCO Object Service Broker IMS TM Client Programs

Using a Session Exit Routine

Each TIBCO Object Service Broker IMS TM client program (**S6BIMxxx**) and the screen I/O continuation program (**S6BDCKRN**) is link-edited with a user-replaceable session exit routine **S6BDCUSX**. You can replace the session exit routine to perform some additional processing to suit local requirements.

Where to Enter the Exit Routine

Enter the session exit routine in:

Phase 1	After the input messages are obtained from the I/O-PCB.
Phase 2	Before the session logs in to the Data Object Broker.
Phase 3	After the session is terminated and before the output messages are inserted to the I/O-PCB.
Phase 4	After the output messages are inserted to the I/O-PCB.

Exit Routine Indicators

S6BDCUSX uses standard OS linkage and is passed a TIBCO Object Service Broker IMS TM Exit Parameter Block mapped by DSECT HDCUSXPB. HDCUSXPB contains the following indicators:

- The exit phase
- The interface style (seamless, conversational, user ID derivation)
- MODname and trancode
- PCB parameter list
- Address of input/output message segments

You can use the session exit routine to provide substitute values for some of these indicators and to bypass some processing in the TIBCO Object Service Broker IMS TM client program or conversational continuation program. Further details are described in HDCUSXPB in the MACRO data set in the TIBCO Object Service Broker distribution library.

Getting Access to IMS TM Data

Overview of the IMS TM Logger Exit

After writing a record to the IMS/DB database, if IMS TM ascertains that it is appropriate according to user-supplied parameters, it calls the Logger Exit routine, DFSFLGX0, if it exists. IMS passes to the exit routine all log data after the data is all written to the IMS log. You use DFSFLGX0 to pass the log data to TIBCO Object Service Broker for processing.

Format 1 DATAIN

Each field is 4 bytes long; COUNT contains the value of 2

COUNT	ADDRESS 1	LENGTH 1	ADDRESS 2	LENGTH 2
-------	-----------	----------	-----------	----------

Function-Specific parameter list

The parameter list is pointed to by ADDRESS1 and the length, 40 bytes, is in LENGTH1

Field Name	Offset	Length	Content
LGWXTYPE	0	1	Call type: 2
LGWXENVR	1	1	Environment type: x'01' = DB/DC online system x'02' = Batch IMS system (includes CICS/DLI) x'03' = Log Recovery utility x'04' = DBCTL system x'05' = DCCTL system
LGWXFLG1	2	1	Flag byte: X'20' 0 = Not /ERE log recovery, 1 = /ERE log recovery x'08' 1 = field LGWTMST exists

Field Name	Offset	Length	Content
	3	1	Reserved
LGWXTOD	4	8	This field has been left here for backward compatibility. The old timestamp format value is in the 00YYDDDF HHMMSSTF format.
LGWXSSID	C	8	IMS subsystem ID.
LGWXBUFR	14	4	Address of the IMS log block data that has been successfully written to the OLDS/SLDS. (This can be a copy of the original IMS buffer.)
LGWXBSIZ	18	4	Length of the log data, in bytes.
LGWXTODN	1C	12	This field contains the current data and time fields, but in the IMS internal packed-decimal format. For further information on the internal packed-decimal time-stamp format, refer to <i>IMS Version 7 Application Programming: Transaction Manager or IMS Version 7 Database Recovery Control (DBRC) Guide and Reference</i> , or subsequent manuals.
LGWXTMST	1C	12	This field contains the current date and time fields, but in the IMS internal packed-decimal format. For further information on the internal packed-decimal time-stamp format, refer to <i>IMS Version 7 Application Programming: Transaction Manager or IMS Version 7 Database Recovery Control (DBRC) Guide and Reference</i> , or subsequent manuals. Refer the LGWXTOD field for the timestamp format used prior to IMS/ESA V6.

IMS Log block data

The block is pointed to by ADDRESS2 and the length is in LENGTH2.

Logger Exit Processing

IMS calls the Logger Exit routine with an initialization call when the logger is opened and with a termination call when the logger is closed. IMS also calls the exit routine and passes log data to it with a write call whenever a block of data is written to the logger.

The Logger Exit routine uses the TIBCO Object Service Broker Call Level Interface to communicate with TIBCO Object Service Broker.

To use this exit routine:

1. Using IEBCOPY, copy the IMS Logger Exit routine S6BFLGX0 and its alias DFSFLGX0 as one entity into either the IMS.SDFSRESL library or another PDS data set concatenated to STEPLIB. This library must be a PDS data set. If the library is a PDSE or the module is not present in STEPLIB, the IMS Transaction Manager control region will not load the exit nor issue any error message.
2. Modify the IMS Transaction Manager control region started task JCL as follows:

- Include the TIBCO Object Service Broker supplied \$HLQNONV\$. \$INSTVER\$.AUTH library in the STEPLIB concatenation in the IMS control region.

```
//STEPLIB DD DISP=SHR,DSN=IMS.SDFSRESL
//          DD DISP=SHR,DSN=customer.pds.containing.DFSFLGX0
...
//          DD DISP=SHR,DSN=$HLQNONV$. $INSTVER$.AUTH
...
```

- Add a //HRNLIB DD statement pointing to the \$HLQNONV\$. \$INSTVER\$.AUTH library:

```
//HRNLIB DD DISP=SHR,DSN=$HLQNONV$. $INSTVER$.AUTH
```

3. Include a DD statement for the HRNIN data set which should contain the following Execution Environment parameters:

```
* This HRNIN is for IMS Logger exit routine, DFSFLGX0
CLIMSGLENMAX=0M,
EXECLOCALSIZE=128K,
EXECSTACKSIZE=128K,
TASKEXECNUM=1,
TASKFILENUM=1,
TASKINITNUM=2,
TASKMISCNUM=1,
```

```

TASKOPERNUM=1,
TASKSORTNUM=1,
* EE PARM
TDS=dob_name,
STANDBY=5,
* Session PARM
U=user_name,                      for example USR40
P=password,
* Transaction options
LIBRARY=library_name,             for example IMSLOGGER
SEARCH=L

```

4. Start up the TIBCO Object Service Broker DOB before bringing up the IMS control region
5. The named library (IMSLOGGER in the example) should contain an entry rule named IMS_LOGGER
6. The Logger exit will pass to the IMS_LOGGER rule a DATAIN area containing 2 format1 blocks of data: the first block is the IMS function-specific parameter list and the second block is the IMS log block. For details refer to [Format 1 DATAIN on page 120](#), and the IBM manual: *IMS Customization Guide* (SC26-9427-05)

Example of Messages

Example 1:

```

DFSFLGX0 - LGXINIT  STARTEE  error, RETCODE=0003, RETDATA=0033
*DFSFLGX0 - Logger Exit Routine disabled
DFSFLGX0 - Reply GO to re-enable Exit Routine when problem is corrected
DFSFLGX0 -Reply END at IMS shutdown to clear this message and end the WTOR task
*30 DFSFLGX0 - Reply GO or END
R 30,GO
*DFSFLGX0 - Logger Exit Routine re-enabled and WTOR task terminated

```

This group of messages indicates:

- The Exit was doing the STARTEE operation and was not successful, in this case the DOB was not up.
- The action taken by the exit was to disable itself.

- The options available were:
 - GO – to re-enable the exit when the problem is corrected (in this case, bring up the DOB)
 - END – if the problem can not be corrected without re-cycling the IMS, then reply END to the WTOR message to end the WTOR task so that IMS can be shutdown gracefully.

Example 2:

```

22:27:00 IMS_LOGGER rule: ENTERING LIB=IMSLOGGER,RULE=IMS_LOGGER !
22:27:00 IMS_LOGGER rule: GOING TO SLEEP FOR 200000 MILLI-SECONDS.
DFSFLGX0 - LGXWRITE CALLRULE error, RETCODE=000D, RETDATA=0000
DFSFLGX0 - CLI request timed out
*DFSFLGX0 -Logger Exit Routine disabled
DFSFLGX0 -Reply GO to re-enable Exit Routine when problem is corrected
DFSFLGX0 -Reply END at IMS shutdown to clear this message and end the
          WTOR task
*31 DFSFLGX0 - Reply GO or END
R 31,END
*DFSFLGX0 - WTOR task terminated

```

This group of messages indicates:

- The two first messages with the time stamp were put out by the test rule, they were not from the Logger Exit.
- The Exit was doing the CALLRULE operation and was timed out, in this case the IMS_LOGGER rule was deliberately in a “wait” and the wait time is longer than the timeout interval set by the CLITIMEOUTLIMIT= value.



Refer to *TIBCO Object Service Broker Parameters* for information about the CLITIMEOUTLIMIT Execution Environment parameter.

- The action taken by the exit was to disable itself and shutdown the Execution Environment.
- The options available were:
 - GO – to re-enable the exit when the problem is corrected.

END – if the problem can not be corrected without re-cycling the IMS, then reply END to the WTOR message to end the WTOR task so that IMS can be shutdown gracefully (illustrated in this case).

Chapter 9

Accessing IMS Via the OTMA Callable Interface

This chapter describes how to access IMS applications via the IMS OTMA callable interface.

Topics

- [Functional Overview, page 126](#)
- [Programming for OTMA, page 127](#)
- [Usage Notes, page 129](#)

Functional Overview

What is the OTMA Callable Interface?

The IMS OTMA Callable Interfaces (C/I) is a function in IMS that provides a high-level interface for access to IMS applications from other z/OS address spaces, including TIBCO Object Service Broker. OTMA consists of API calls that you can use to:

1. Join the IMS/OTMA XCF group
2. Connect to IMS
3. Allocate a communications session
4. Send IMS transactions names and commands
5. Receive output from IMS
6. Close the communications session
7. Leave the XCF group

The TIBCO Object Service Broker OTMA Callable Interface supports IMS OTMA. TIBCO Object Service Broker access is implemented using the TIBCO Object Service Broker OTMA tools \$OTMA and @OTMA_MAP, which is a system interpreted table. For details about the use of these tools refer to TIBCO Object Service Broker *Shareable Tools*.

See Also The IBM manual, *IMS Open Transaction Manager Access Guide and Reference*, SC26-9434.

Programming for OTMA

Requirements

To communicate with IMS using TIBCO Object Service Broker OTMA, you need the following:

- The \$OTMA tool
- The OTMA interface control block mapped by MAP table @OTMA_MAP.
- You must refresh the contents of this control block mapping by issuing a

```
GET @OTMA_MAP(address_of_@OTMA_MAP)
```

request after each use of the \$OTMA tool. All parameters and data passed to OTMA must be set by updating this control block before calling the \$OTMA tool.

- Other user-defined MAP tables that map the RECEIVE and possibly the SEND buffers used to transfer data between TIBCO Object Service Broker and OTMA.
- You have to consult your IMS application documentation to understand the data format to be used in running an IMS transaction via OTMA and to map the output returned to your OTMA client.

Sample Rules Provided

To understand the elements of a basic TIBCO Object Service Broker application that uses OTMA, you can use some of the sample rules in the COMMON library. These rules execute the IMS TM PART installation verification transaction. Refer to the *IBM Installation Volume 1: Installation and Verification* manual, in the “IMS Sample Application” section, to find details of the PART Message Processing Program (MPP) transaction.

The main rule is called

```
SAMPLE_OTMA_CALL
```

and executes the following OTMA steps by calling other rules:

1. Allocate SESSION storage for the OTMA interface control block using the @MAP tool
2. Issue OTMA OPEN to connect with IMS and identify the XCF group, client member name, OTMA server name and other attributes necessary to perform the otma_open operation.

3. Issue OTMA ALLO to allocate an OTMA session to exchange messages.

This request issues the `otma_allocate` operation and you use it to specify the transaction or command to run, the synchronization level you want to use, and the RACF group name to be used for OTMA transactions and commands.

4. Allocate SESSION storage for the OTMA SEND and RECEIVE buffers using the `@MAP` tool.
5. Set the transaction or command data into the SEND buffer and issue the OTMA SEND request to transfer the request to the OTMA server via an `otma_send_receive` interface call. Data is returned in your RECEIVE buffer.

You must set various fields in `@OTMA_MAP` to define: the length and address of your SEND and RECEIVE buffers and the required transaction or command data in your SEND buffer.

6. Map the returned data via your user-defined RECEIVE MAP table. Issue a

```
GET user_RECEIVE_TBL(address_of_RECEIVE_buf)
```

to cause your MAP table to reflect the data returned via your OTMA RECEIVE buffer.

At this point, you can display or process the data in your OTMA client application. If no further transactions or IMS commands are to be run, you can dismantle your OTMA structure by:

1. Issuing an OTMA FREE request to free the OTMA session via an `otma_free` request. This releases the session created by the OTMA ALLO request.
2. Issuing an OTMA CLOS request to leave the XCF group and free up storage used for communication with OTMA. This invokes the `otma_close` interface call.

The SESSION storage allocated for SEND, RECEIVE, and `@OTMA_MAP` MAP tables are freed automatically when the client OTMA session is terminated.

Session Termination

After each call of the `$OTMA` interface tool, check the return codes in `@OTMA_MAP` to ensure that the operation completed successfully. If you decide to terminate the session, perhaps because of an error, release the OTMA resources in the reverse order they were acquired. The sample rules `@OTMA_CHKRETRSN` and `@OTMASIGNAL` attempt to implement this recovery by signalling various failures back to the invoking rule's "ON condition" routine.

Usage Notes

z/OS and IMS System Requirements

To use IMS OTMA, your system must meet certain minimum requirements such as the IMS version, IMS startup parameters, XCF group name, OTMA initialization after a z/OS IPL, and RACF resource definition. Check with your systems programmer and systems administrator for details and set up.

Invoking the Tool and the System Map Table

The interface between the rules language and IMS OTMA is controlled by the \$OTMA tool and its corresponding system interpreted table @OTMA_MAP.

Interpreter TCB Held for Communications

The communications between the TIBCO Object Service Broker OTMA Callable Interface and IMS OTMA is synchronous, therefore an interpreter TCB is held while waiting for a response.

Error Handling

The return code and reason codes are stored in the map table; refer to @OTMA_MAP in *TIBCO Object Service Broker Shareable Tools* for details. You can check these codes in your rules. You can find a complete description of the return codes and reason codes in the IBM manual, *IMS Open Transaction Manager Access Guide and Reference*, SC26-9434.

Example Rules and Tables

Sample rules and tables are available that you can use as a template:

- The rule names start with @OTMA and are in the COMMON library
- The table names start with @OTMA, under UNIT=OTMA

Chapter 10 **Accessing External Routines**

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

Topics

- [Functional Overview, page 132](#)
- [Observing Standard Conventions, page 134](#)
- [Making a COBOL Program Compatible with TIBCO Object Service Broker, page 138](#)
- [Making a PL/I Program Compatible with TIBCO Object Service Broker on page 143](#)
- [Making a C Program Compatible with TIBCO Object Service Broker, page 146](#)
- [Identifying Your External Routine to TIBCO Object Service Broker, page 152](#)
- [Calling the Routine, page 157](#)

Functional Overview

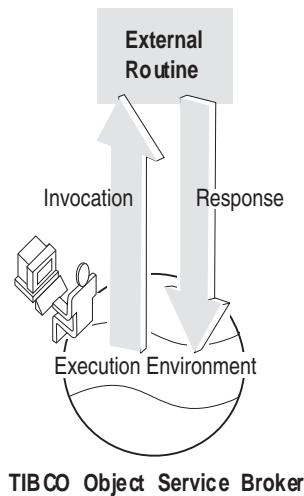
How Does TIBCO Object Service Broker Process an External Routine?

You can pass 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 routine and waits for control to be passed back to it when the external routine is finished.
3. The link to the external routine is deleted when the transaction ends.

Process Flow

This illustration shows the process flow between TIBCO Object Service Broker and an external routine.



Steps Required to Use an External Routine

The major steps in preparing and executing an external routine are:

1. Code, compile, and link the external routine.

Routines should be re-entrant where possible. This is not a requirement but it saves storage in a multiple-session system. Also, these routines should use standard assembler linkage conventions.

2. Identify the external routine and its characteristics to TIBCO Object Service Broker.
3. Prepare the location where the external routine is to execute.

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

Transaction Level of the Routine

TIBCO Object Service Broker treats all external routines in the following way: during a transaction, the routine is loaded once per transaction level, TIBCO Object Service Broker branches and links to the routine whenever it is called, and deletes it when the transaction ends.

For example, if you execute the ABC rule from the workbench, and this rule calls an external routine, the external routine is loaded for transaction level 0. The same rule can access table XYZ, invoking a trigger rule in the process. If the trigger rule calls an external routine, the routine is loaded for transaction level 1. This all takes place in the same transaction but at different levels.

Cleanup of System Service Requests

If an external routine requests system services, it should perform any necessary cleanup (for example, issuing FREEMAIN macros to release storage obtained by the GETMAIN macros). Code these requests so that they operate independently of the environment (TSO, CICS, IMS, and so on) where they execute.

Error Handling

An external routine can override TIBCO Object Service Broker error-handling. It can have its own STAE/ESTAE macro to handle abnormal terminations and SPIE/ESPIE to handle program interrupts. If it does, it must restore TIBCO Object Service Broker error-handling, including the program mask, before returning. Failure to do so causes unpredictable results.

See Also *TIBCO Object Service Broker Programming in Rules* about TIBCO Object Service Broker transaction processing.

Observing Standard Conventions

TIBCO Object Service Broker executes as an assembler environment; you must write and link your program so it can be called by an assembler program using standard conventions as outlined below.

Information Available to an External Routine

When an external routine receives control, the following information is available:

Register	Contents
1	The address of an argument list: Each address in the argument list is a full word. The last address has its left-most bit set. If the routine is a function, the <i>first</i> address points to the area for the returned value.
13	The address of an 18-word save area.
14	The return address of the calling routine.
15	The address of the called routine.

Use of the AMODE and RMODE Attributes

TIBCO Object Service Broker uses the AMODE and RMODE attributes of the load module (for an external routine) to determine:

- Whether the module is loaded below the line
- Whether the routine arguments must be built below the line
- Which addressing mode is in effect when the routine receives control

When control is passed back to TIBCO Object Service Broker, you can access a return code in register 15 by using the [RETURN_CODE](#) tool.

Storage Requirements

This table shows how TIBCO Object Service Broker syntax is mapped to assembler storage requirements.

TIBCO Object Service Broker			Assembler Storage Requirements
Syntax	Length	Decimal	
C	<i>n</i>		CL <i>n</i>
V	<i>n</i>		CL <i>n</i> +2
P	<i>n</i>	<i>d</i>	PL <i>n</i>
B	<i>n</i>		XL <i>n</i>
F	4		E
F	8		D
F ^a	16		Not available
RD ^b	<i>n</i>		XL <i>n</i>
UN	<i>n</i>		<i>n</i> /2CU

a. TIBCO Object Service Broker supports floating point, 16 bytes long, which is not supported in COBOL and therefore cannot be used in COBOL subroutines.

b. An RD field consists of a four-byte non-exclusive binary length followed by the data.



TIBCO Object Service Broker date (D) semantic type is not converted; it is returned as a binary number.

Example Assembler Program

The following external routine returns the number of occurrences of a specified character in a given string. The routine requires three arguments: *STRING*, *CHAR*, and *FOLD*. It returns a value through *COUNT*. The routine and its arguments must be identified to TIBCO Object Service Broker as described in [Identifying Your External Routine to TIBCO Object Service Broker on page 152](#).

```
*****
*          CHRINSTR(STRING,CHAR,FOLD)                      *
*                                                         *
*          RETURNS THE NUMBER OF OCCURRENCES IN STRING OF THE SINGLE *
*          CHARACTER CHAR;                                     *
*          IF FOLD IS 'Y' IGNORE CASE-SENSITIVITY;          *
*                                                         *
*          SPACE 2
PARMLIST DSECT
RCOUNT   DS      A                      ADDRESS OF RETURNED COUNT
*                                     ... (C,P,4,0)
PSTRING  DS      A                      ADDRESS OF INPUT STRING
*                                     ... (S,V,256,0)
PCHAR    DS      A                      ADDRESS OF INPUT CHAR
*                                     ... (S,V,1,0)
PFOLD    DS      A                      ADDRESS OF INPUT FOLD
*                                     ... (S,C,1,0)
*          SPACE 2
CHRINSTR AMODE 31                      EXECUTES IN 31-BIT ADDRESSING
CHRINSTR RMODE ANY                     RESIDES ANYWHERE
CHRINSTR CSECT
*
*          ESTABLISH ADDRESSABILITY
*
*          USING CHRINSTR,R12
*          USING PARMLIST,R11
*          STM   R14,R12,12(R13)        SAVE CALLER'S REGISTERS
*          LR    R12,R15                CSECT BASE REGISTER
*          LR    R11,R1                 ARGUMENT BASE REGISTER
*
*          LOAD THE INPUT STRING AND CHARACTER
*
*          L      R2,PSTRING             POINT TO STRING ARGUMENT
*          LH     R3,0(R2)               LOAD LENGTH
*          LA     R2,2(R2)               SKIP PREFIX LENGTH
*          XR     R4,R4                  RESET COUNTER
*          LTR    R3,R3                  EMPTY STRING?
*          BZ     RETURN                 YES, RETURN RESULT
*          L      R5,PCHAR               POINT TO CHAR ARGUMENT
*          LA     R5,2(R5)               SKIP PREFIX LENGTH
*
*          PROCESS FOLD INDICATOR
*
*          L      R6,PFOLD               POINT TO FOLD ARGUMENT
*          CLI    0(R6),C'Y'             SHOULD WE FOLD?
*          BNE    LOOP                   NO, CONTINUE
*
```

```

*          CODE TO IGNORE DIFFERENCE OF CASE IF FOLD IS 'Y' CAN BE PUT HERE
*
*          MAIN LOOP: PROCESS ONE CHARACTER AT A TIME
*
LOOP      DS      0H
          CLC      0(1,R2),0(R5)          CURRENT CHAR MATCH?
          BNE      NEXTCHAR              NO, CONTINUE
          LA       R4,1(,R4)              ELSE, INCREMENT COUNTER
NEXTCHAR  DS      0H
          LA       R2,1(,R2)              ADVANCE TO NEXT CHARACTER
          BCT      R3,LOOP                LOOP IF NOT END OF STRING
*
*          COPY RESULT TO RETURN AREA
*
RETURN    DS      0H
          L        R2,RCOUNT              POINT TO RESULT AREA
          CVD      R4,PRESULT              CONVERT RESULT TO DECIMAL
          ZAP      0(4,R2),PRESULT+4(4)    COPY TO RESULT AREA
          LM       R14,R12,12(R13)          RESTORE CALLER'S REGISTERS
          XR       R15,R15                 ZERO RETURN CODE
          BR       R14                     RETURN TO CALLER
*
*          DATA AREA
*
          DS      0D                      ALIGN ON DOUBLE-WORD BOUNDARY
PRESULT   DS      PL8                     AREA FOR CONVERSION
          LTORG   ,
          COPY    REGEQU                  REGISTER DEFINITIONS
          END     CHRINSTR

```

See Also *TIBCO Object Service Broker Programming in Rules* about TIBCO Object Service Broker syntax.

TIBCO Object Service Broker Shareable Tools about the tools.

Making a COBOL Program Compatible with TIBCO Object Service Broker

Requirements

Use IBM Enterprise COBOL for z/OS or later compiler if the external routine is to be run on multiple threads of a multi-threaded application, such as a Native Execution Environment. Older programs written in OS/VS Cobol or COBOL II should be recompiled with this new compiler.

COBOL Run-Units

A COBOL run-unit is created when you invoke a COBOL external routine and terminated when the COBOL program issues a GOBACK to return control to TIBCO Object Service Broker. A subsequent call to the routine within the same transaction causes the routine to be executed as though it were being called for the first time. If the routine opens files or sets flags, expecting to use this environment on subsequent calls, processing errors occur, with possible abends or incorrect logic paths.

How to Prevent Premature Termination of the COBOL Run-Unit

To ensure that a COBOL run-unit is not terminated prematurely and is active for the life of the current rule stream/transaction level, you can take the following steps. Additional information can be found in the IBM “Enterprise COBOL for z/OS Programming Guide”.

1. Define the COBOL routine to TIBCO Object Service Broker in the ROUTINES table as LANGUAGE=LEPERSIST. This creates a CEEPIPI environment for the COBOL run unit/Language Environment enclave.
2. Specify the COBOL compiler THREAD option for the compilation. (The module will also execute successfully in a non-THREAD environment with this option.)
3. Specify the PROGRAM-ID RECURSIVE option in your COBOL source code.
4. Linkedit the COBOL module with RENT option. Do not include any IGZEOPT definition because this facility is not supported with the THREAD option.

A sample COBOL external routine is included in the COBOL sample library and is called COBENTXR.

Link-Edit and Runtime Options

Your COBOL external routine can override TIBCO Object Service Broker error-handling with a runtime option. If the program has its own error handling, it must restore TIBCO Object Service Broker error-handling, including the program mask, before returning. Failure to do so produces unpredictable results.

Syntax Mapping

The following table shows how TIBCO Object Service Broker syntax is mapped to COBOL syntax.

TIBCO Object Service Broker			
Syntax	Length	Decimal	COBOL Syntax Declaration #
C	<i>n</i>		PIC X(<i>n</i>)
V	<i>n</i>		PIC 9(4) USAGE COMP-4 and PIC X(<i>n</i>) ^a
P	<i>n</i>	<i>d</i>	PIC S9(2 <i>n</i> -1- <i>d</i>)V9(<i>d</i>) USAGE COMP-3
B	<i>n</i> = 2		PIC S9(4) USAGE COMP-4
B	<i>n</i> = 4		PIC S9(9) USAGE COMP-4
F	<i>n</i> = 4		COMP-1
F	<i>n</i> = 8		COMP-2
F	<i>n</i> = 16		Not available ^b
RD ^c	<i>n</i>		PIC X(<i>n</i>)
UN	<i>n</i>		PIC N(<i>n</i> /2) USAGE NATIONAL

- a. The first part of the COBOL syntax for V describes the length of the argument and the second part contains the data. Refer to [Example COBOL Program on page 141](#) for an example of this usage.
- b. TIBCO Object Service Broker supports floating point, 16 bytes long, which is not supported in COBOL and therefore cannot be used in COBOL subroutines.
- c. An RD field consists of a four-byte non-exclusive binary length followed by the data.



TIBCO Object Service Broker date (D) semantic type is not converted; it is returned as a binary number.

Example COBOL Program

The following routine shows how to define the arguments in COBOL. The sample external routine requires five arguments. The routine and its arguments must be identified to TIBCO Object Service Broker as described in [Identifying Your External Routine to TIBCO Object Service Broker on page 152](#).

```

CBL MAP,RENT,NOSEQUENCE,TEST(SYM),THREAD,VBREF
IDENTIFICATION DIVISION.
PROGRAM-ID. USEARG RECURSIVE.
AUTHOR. JANET JONES.
INSTALLATION. SITE.
DATE-WRITTEN. 21/3/2007.
*****
*
*      USEARG - SAMPLE PROGRAM:
*
*      N.B.: CALLED FROM OSB WITH PARAMETERS;
*      CALL IS BY REFERENCE
*
*      OSB ARGUMENTS(USEARG) TABLE CONTAINS THE FOLLOWING
*      DEFINITION FOR PARAMETERS
*
*      B2: C B 2 0
*      B4: C B 4 0
*      C16: S C 16 0
*      P5D2: Q P 5 2
*      V128: S V 128 0
*
*****
ENVIRONMENT DIVISION.
*
DATA DIVISION.
*
WORKING-STORAGE SECTION.
*
LINKAGE SECTION.
01 B2 PIC S9(4) USAGE COMP-4.
01 B4 PIC S9(9) USAGE COMP-4.
01 C16 PIC X(16).
01 P5D2 PIC 9(7)V9(2) USAGE COMP-3.
01 V128.
*   FOR "V" SYNTAX PREFIX DATA WITH ACTUAL LENGTH OF DATA. (DOES
*   NOT INCLUDE LENGTH OF ITSELF.)
05 V128-LENGTH PIC 9(4) USAGE COMP-4.
05 V128-TEXT PIC X(128).
*
PROCEDURE DIVISION USING B2, B4, C16, P5D2, V128.
*
*      YOUR CODE
*
      MOVE 12 TO B2.
      MOVE 605 TO B4.
      MOVE 'SUCCESS' TO C16.

```

```
MOVE 311.73 TO P5D2.  
MOVE 12 TO V128-LENGTH.  
MOVE 'SOME MESSAGE' TO V128-TEXT.  
GOBACK.
```

See Also *TIBCO Object Service Broker Programming in Rules* about TIBCO Object Service Broker syntax.

Making a PL/I Program Compatible with TIBCO Object Service Broker

Requirements

You can write a PL/I program as a callable routine in TIBCO Object Service Broker but you cannot write your PL/I program as a function (that is, you cannot write your PL/I program to return a value unless you pass the value back through an argument).

Link-Edit Options

Link-edit a PL/I program with option ENTRY PLICALLA.

Syntax Mapping

The following table shows how TIBCO Object Service Broker syntax is mapped to PL/I syntax.

TIBCO Object Service Broker			PL/I Syntax Declaration #
Syntax	Length	Decimal	
C	<i>n</i>		CHAR(<i>n</i>)
V	<i>n</i>		CHAR(<i>n</i>) VARYING
P	<i>n</i>	d	DECIMAL FIXED(2 <i>n</i> -1,d)
B	<i>n</i>		BINARY FIXED(8 <i>n</i> -1)
F	<i>n</i> = 4		BINARY FLOAT(21)
F	<i>n</i> = 8		BINARY FLOAT(53)
F	<i>n</i> = 16		BINARY FLOAT(109)
RD ^a	n		CHAR(n) VARYING
UN	n		WIDECHAR(n/2)

a. An RD field consists of a 4-byte non-exclusive binary length followed by the data.



TIBCO Object Service Broker date (D) semantic type is not converted; it is returned as a binary number.

Example PL/I Program

The external routine in the following example takes a number in the first argument and returns the logarithm of the number in the second argument. The routine and its arguments must be identified to TIBCO Object Service Broker as described in [Identifying Your External Routine to TIBCO Object Service Broker on page 152](#).

```
plilog: proc(number,result) options(main);
  dcl (number,result) binary float (21);
  dcl log10 builtin;
  result= log10(number);
end plilog;
```

External PL/I Routine to Concatenate Strings for TIBCO Object Service Broker

The external routine in the following example concatenates a string to itself a specified number of times. To return a value in a non-numeric argument, the PL/I program must use a pointer to access the argument; therefore, STRING points to the PSTRING argument. The routine and its arguments must be identified to TIBCO Object Service Broker as described in [Identifying Your External Routine to TIBCO Object Service Broker on page 152](#).

```

/* PLISTR - EXAMPLE PROGRAM */
/* OSB ROUTINES table contains the following: */
/* PLISTR PL/I N 0 0 */
/* OSB ARGUMENTS(PLISTR) table contains the following: */
/* 1 PSTRING Y S C 128 */
/* 2 PLENGTH N Q B 2 */
plistr: proc(pstring,length) options(main);
    dcl (pstring) fixed bin(15)
    dcl (length) fixed bin(15);
    dcl string char(128) based(pl),
        pl pointer init(addr(pstring));
    string= repeat('xyz ',length);
end plistr;

```

See Also *TIBCO Object Service Broker Programming in Rules* about TIBCO Object Service Broker syntax.

Making a C Program Compatible with TIBCO Object Service Broker

Requirements

Your C program should be written and receive arguments according to standard C coding conventions.

Syntax Mapping

The following table shows how TIBCO Object Service Broker syntax is mapped to C syntax.

TIBCO Object Service Broker			C Syntax Declaration
Syntax	Length	Decimal	
C	n		char [n]
V	n		short; char [n] ^a
P	n	d	decimal [n,d] ^b
B	$n = 2$		short
B	$n = 4$		long
F	$n = 4$		float ^b
F	$n = 8$		double ^b
F	$n = 16$		long double ^b
RD	n		unsigned char[n] ^c
UN	n		unsigned char[n]

a. The first part of the C syntax for V describes the length of the argument and the second part contains the data.

b. TIBCO Object Service Broker supports packed decimal and IBM hexadecimal floating point. The C declarations are valid only for IBM's mainframe C compiler.

c. An RD field consists of a four-byte inclusive binary length followed by the data.



TIBCO Object Service Broker date (D) semantic type is not converted; it is returned as a binary number.

See Also

TIBCO Object Service Broker Programming in Rules about TIBCO Object Service Broker syntax.

Sample

C Program

The following routine shows how to define the arguments in C. The routine must be identified to TIBCO Object Service Broker as described in [Identifying Your External Routine to TIBCO Object Service Broker on page 152](#). In the ROUTINES table, for routines written in IBM C, the Language specified in the ROUTINES table should be LEPERSIST.

```
/*    dayXmasC - concatenate a number to a string    */

static void join(
    char * * ptptr,
    char * sptr
)
{
    char c;
    char * tptr = *ptptr;

    do {
        *tptr++ = c = *sptr++;
    } while ( c );
    *ptptr = tptr;        /* update pointer for caller */
    *(tptr-1) = ' ';      /* change null to a blank */
}

void DAYXMAS(
    char * result,
    int * count,
    char * item
)
{
    int len;
    char *p, *p0;
    short *p2;
    char temp[80];

    static char * numbers[14] = { "no", "a", "two", "three", "four", "five",
        "six", "seven", "eight", "nine", "ten", "eleven", "twelve", "many" };
    static char * minus = "minus";

    int itemlen = (short) *((short *) item);
    int i = *count;
    p = p0 = result + 2;
    p2 = (short *) result;

    /* Make a C-format string from the input string. */
    for (len=0; len<itemlen; len++)
        temp[len] = item[len+2];
    temp[itemlen] = 0;
}
```

```
/* Adjust the count so that it is between 0 and 13 inclusive. */
if ( i < 0 ) {
    join(&p,minus);
    i = -i;
}
if ( i > 12 ) i = 13;

join(&p,numbers[i]);
join(&p,temp);
len = (int) (p-p0-1);    /* length of output string */
*p2 = (short) len;
}
```

Table ROUTINES

The table ROUTINES should contain:

EDITING TABLE	: ROUTINES
TABLE TYPE	: TDS
COMMAND ==>	

NAME	: DAYOFCHRISTMAS
LANGUAGE	: LEPERSIST
FUNCTION	: Y
TYPE	:
SYNTAX	: V
LENGTH	: 80
DECIMAL	: 0
LOADNAME	: DAYXMAS
SCOPE	:
NODENAME	:
	:
	:
LIBNAME	:
	:
	:

Table ARGUMENTS(DAYOFCHRISTMAS)

The table ARGUMENT(DAYOFCHRISTMAS) should contain:

BROWSING TABLE : ARGUMENTS(DAYOFCHRISTMAS)
COMMAND ==>

NUMBER	NAME	INOUT	TYPE	SYNTAX	LENGTH	DECIMAL
1	COUNT	N	C	B	4	0
2	ITEM	N	S	V	32	0

Rule That Calls the C Program

The first rule calls the second one, which calls the C routine.

RULE EDITOR ==>	SCROLL: P
TWELVE;	
CALL XDAY(12, 'drummers drumming');	1
CALL XDAY(11, 'lords a-leaping');	2
CALL XDAY(10, 'pipers piping');	3
CALL XDAY(9, 'ladies dancing');	4
CALL XDAY(8, 'maids a-milking');	5
CALL XDAY(7, 'swans a-swimming');	6
CALL XDAY(6, 'geese a-laying');	7
CALL XDAY(5, 'gold rings');	8
CALL XDAY(4, 'calling birds');	9
CALL XDAY(3, 'French hens');	A
CALL XDAY(2, 'turtle doves');	B
CALL XDAY(1, 'partridge in a pear tree');	C
PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE	

```

      RULE EDITOR ==>                                SCROLL: P
XDAY(N, X);
-
- -----
- -----+-----
- CALL MSGLOG(DAYOFCHRISTMAS(N, X));                | 1
- -----

```

PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE

JCL to compile and link the sample

```

//CMP6000 EXEC PROC=EDCCB,
// CPARM='RENT,SHOW,SO,XREF',
// INFILE='S6B.TST.C(DAYXMAS)',
// OUTFILE='S6B.TST.LOAD(DAYXMAS),DISP=SHR'
//BIND.SYSIN DD *
  ENTRY DAYXMAS
  NAME DAYXMAS(R)
/*

```

Identifying Your External Routine to TIBCO Object Service Broker

Specify the Table Entries

You identify the external routine and its load module name to TIBCO Object Service Broker through an entry in the ROUTINES table. If the routine has arguments, specify these in a table instance of the ARGUMENTS table. Ensure that you have adequate security authorization 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 screen shows an extract of the ROUTINES table. Scroll right using PF11 to see additional fields.

EDITING TABLE : ROUTINES							
COMMAND ==>							
		SCROLL: P					
NAME	LANGUAGE	FUNCTION	TYPE	SYNTAX	LENGTH	DECIMAL	
-----	-----	-	-	-	-----	-----	
- ABEND	ASSEMBLER	N			0	0	
- ADMCHART	ASSEMBLER	N			0	0	
- ASREAD	ASSEMBLER	N			0	0	
- BINARY_TO_LOGIC	ASSEMBLER	Y	L	C	1	0	
- BTOPACKD	ASSEMBLER	Y		C	1	0	
- CATROW	ASSEMBLER	N			0	0	
- CCOB11	HLLCOBOL	N			0	0	
- CCOB12A	HLLCOBOL	N			0	0	
- CCOB12B	HLLCOBOL	N			0	0	
- CDIR500	HLLCOBOL	N			0	0	
- CDIS510	HLLCOBOL	N			0	0	
- CFOR500	HLLCOBOL	N			0	0	
- CGET000	HLLCOBOL	N			0	0	
- CGET500	HLLCOBOL	N			0	0	
- CHHATT	ASSEMBLER	N			0	0	
- CCHHEAD	ASSEMBLER	N			0	0	
- CHHIST	ASSEMBLER	N			0	0	
PFKEYS: 4=INSERT 16=DELETE 5=FIND NEXT 6=CHG NEXT 18=EXCLUDE 3=SAVE 12=CANCEL							

Type the following information in the fields of the ROUTINES table for your external routine:

NAME	The name used to invoke the external routine. If this name is different from the name of the load module, see the explanation for the LOADNAME field.
LANGUAGE	The language in which the external routine is written, mainly for documentation purposes, with the exception of the following keywords:
CICS	Indicates that this is a CICS program to be run under the CICS task in a CICS Execution Environment. For an external routine with OS linkage that is to be run under a TIBCO Object Service Broker task in a CICS Execution Environment, specify the language used, not the value CICS.
COBOL	For HLI routines.
LE	Indicates that this routine is to be run in a pre-initialized Language Environment (LE) enclave, the CEEPIPI environment. This routine is a stand-alone routine that does not refer to global variables. For LANGUAGE=LE, you must provide the appropriate CEEUOPT in your LE-compliant external routines, because the CEEPIPI enclave is initialized according to the specification of the first external routine called. You should use the same CEEUOPT options for all external routines in the same class for the same TIBCO Object Service Broker Execution Environment, because the same CEEPIPI enclave cannot support different runtime options.

LEPERSIST Indicates that this routine is to be run in a pre-initialized LE enclave, the CEEPIPI environment. This routine is one of a set of routines that refer to global variables used by other routines in the set.

For LANGUAGE=LEPERSIST, you must provide the appropriate CEEUOPT in your LE-compliant external routines, because the CEEPIPI enclave is initialized according to the specification of the first external routine set being called. You should use the same CEEUOPT options for all external routines in the routine set, because the same CEEPIPI enclave cannot support different runtime options after it is initialized. You can use different CEEUOPT options for different routine sets, because each set triggers a re-initialization of the LE enclave. The routine set boundary is defined by the first call to the routine in the set and the end of that stream level.

For LANGUAGE values other than LE or LEPERSIST, the external routines are run outside the CEEPIPI environment. Each time an LE-compliant external routine is called, a fresh LE enclave is initialized according to the specification of that external routine. Therefore this class of external routines can have different runtime options within the same TIBCO Object Service Broker Execution Environment. It is your responsibility to provide the appropriate CEEUOPT in your external routines.

FUNCTION	Specifies whether the external routine returns a value (Y or N).
TYPE	If the external routine is a function, the TIBCO Object Service Broker semantic type of the value returned.
SYNTAX	If the external routine is a function, the TIBCO Object Service Broker syntax of the value returned.
LENGTH	If the external routine is a function, the length of the value returned.

DECIMAL	If the external routine is a function that returns a value with digits to the right of the decimal, the number of digits.
LOADNAME	The name of the load module in the external routines load library. You can leave this blank if the name of the load module is the same as the routine name (that is, in the NAME field). You must scroll right to see this field.
SCOPE	Not used for z/OS
NODENAME	Not used for z/OS
LIBNAME	Not used for z/OS

Adding an Entry to the ARGUMENTS Table

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

The following arguments are for the assembler program shown in [Example Assembler Program on page 136](#).

EDITING TABLE : ARGUMENTS(CHRINSTR)

COMMAND ==>

SCROLL: P

NUMBER	NAME	INOUT	TYPE	SYNTAX	LENGTH	DECIMAL
1	COUNT	Y	C	P	8	
2	STRING	N	S	V	256	
3	CHAR	N	S	V	1	
4	FOLD	N	L	C	1	

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

Type appropriate information in the fields of the ARGUMENTS table:

NUMBER	Position of the argument in the argument list. The positions must be sequential starting at 1.
NAME	Argument name
INOUT	Specifies whether the value of the argument can be changed by the external routine. If so, 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 and N.
TYPE	Argument semantic type
SYNTAX	Argument syntax
LENGTH	Argument length
DECIMAL	Number of digits to the right of the decimal, if any

See Also *TIBCO Object Service Broker Managing Data* about the Table Editor
TIBCO Object Service Broker Programming in Rules about syntax and semantic data types

Calling the Routine

Put the Routine in a Load Library

Before you can call the routine from within TIBCO Object Service Broker you must compile, link-edit, and place the routine in a load library:

- If the routine is to run in a CICS environment and the language specified in the **Language** field of the ROUTINES table is set to CICS, the routine must be placed in the CICS DFHRPL library. Refer to [Add an Entry in the ROUTINES Table on page 152](#) for information about the ROUTINES table.
- In all other cases, place the load module into a library concatenated to the DD statement HRNEXTR.

Call the Routine From TIBCO Object Service Broker

When the executable code is in an external routine load library, and you made the appropriate entries in the ROUTINES and ARGUMENTS tables, you can call the routine from a TIBCO Object Service Broker rule.

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);

How are Exceptions Handled?

No exceptions are trapped by TIBCO Object Service Broker during external routine invocation and execution, for example if the external routine is not found or it fails. However, if a failure occurs, a message is returned to the message log.

See Also *TIBCO Object Service Broker Programming in Rules* about the rules language and writing rules.

Chapter 11 **Using User Builtin Routines**

This chapter describes how to use user builtin routines.

Topics

- [Functional Overview, page 160](#)
- [Programming Considerations, page 161](#)
- [Sample User Builtin Routines, page 163](#)

Functional Overview

What are User Builtin Routines?

User builtin routines are user-written assembler routines that are similar to external routines. They can be packaged and link-edited directly into TIBCO Object Service Broker, unlike external routines, which are documented in [Chapter 10, Accessing External Routines, on page 131](#).

User builtin routine must be installed following the procedure in *TIBCO Object Service Broker for z/OS Installing and Operating*.

What Are the Requirements for User Builtin Routines?

- A user builtin routine must meet the following requirements:
- Be written in z/OS assembler
 - Be fully re-entrant with AMODE31 and RMODE ANY
 - Use standard assembler linkage conventions
 - Receive and update parameters, and return values as documented in [Chapter 10, Accessing External Routines, on page 131](#)
 - Set one of the following values in Register 15 at the end of the routine before returning control to TIBCO Object Service Broker:

0	The routine was successful.
8	The routine failed. This also causes the calling rule to fail.

Failure to set Register 15 to 0 or 8 causes unpredictable abends.

Available Examples

Refer to [, Samples Available, on page 163](#) and the USRBLTIN member of the ASM data set provided with TIBCO Object Service Broker for examples of user builtin routines.

Programming Considerations

Acquiring and Releasing Storage

If your routine calls another program or requires working storage, in most cases use the z/OS GETMAIN and FREEMAIN macros to acquire and release the storage. In some high activity programs, GETMAIN and FREEMAIN can have significant overhead. In these cases, use the \$SAVE macro to make use of storage in the TIBCO Object Service Broker transaction save stack.

Using the \$SAVE Macro

Syntax

The syntax of the \$SAVE macro is as follows:
label \$SAVE length,BASEREG=reg
where:

<i>label</i>	The entry point name of the routine
<i>length</i>	The number of bytes of working storage required. This is in addition to the standard 80 bytes required for a 72 byte save area, followed by 8 bytes for stack control pointers.
<i>reg</i>	The base register to be used for the program. The default is R12.

Requirements

- If you use \$SAVE, you must:
- Use R13 as the base register for this storage
 - Use the first 72 bytes as a normal save area
 - Not use the next 8 bytes as these are used by \$SAVE
 - Copy the WALIST copy book

The RETURN macro or other standard linkage conventions can be used to return to the caller of the routine.

Available Examples

Refer to the USRWTO sample program in [“Samples Available”](#) on page 163 and the USRBLTIN member of the ASM data set provided with TIBCO Object Service Broker for examples of using \$SAVE.

\$SAVE Macro Storage Usage

The \$SAVE macro does the following:

- Acquire storage of length *length+80* and point Register 13 to it.
- Save all the registers in the save area of the caller.
- Chain the save area pointer between the caller and the \$SAVE storage.
- Set the base register.

Sample User Builtin Routines

Samples Available

The USRBLTIN member in the ASM data set includes these sample routines:

- USRSLEEP
- USRTUM
- USRWTO

The following section describes the entries required for USRSLEEP, and provides a sample rule showing how to call it.

USRSLEEP

This routine has one argument, TIME of *nn* hundredths of a second.

USRBLTIN Entry

\$ROUTINE NAME=SLEEP,MODULE=USRSLEEP,	X
ARGS=((,B,4,0)),	X
ARGNAMES=(TIME)	

USRCSECT Entry

```
TITLE 'A PROGRAM TO SLEEP FOR N 100THS OF A SECOND.'
* * * * *
*
* WARNING: THIS ROUTINE WILL PUT THE INTERPRETER TASK INTO A WAIT.
* IT SHOULD THEREFORE NOT BE USED IN A MULTI-USER EE SUCH AS CICS,
* IMS, OR NATIVE. THIS ROUTINE IS SUITABLE ONLY FOR BATCH & TSO.
*
* THIS PROGRAM SLEEPS FOR A TIME SPECIFIED IN ARGUMENT1, WHICH IS
* SPECIFIED IN 100THS OF A SECOND.
*
* THIS PROGRAM CAN EXECUTE AS AN OSB EXTERNAL ROUTINE AS DEFINED IN
* THE EXTERNAL ENVIRONMENTS GUIDE; HOWEVER, THE IMPLEMENTATION SHOWN
* HERE IS TO EXECUTE AS A USER DEFINED BUILTIN ROUTINE AS DOCUMENTED
* IN THE TIBCO® OBJECT SERVICE BROKER FOR Z/OS INSTALLING AND
* OPERATING MANUAL.
*
* ASSEMBLE & LINK TO HRNEXTR LOAD LIBRARY IF RUNNING AS AN EXTERNAL
* ROUTINE. IF RUNNING AS A USER DEFINED BUILTIN ROUTINE, IMPLEMENT
* WITH JCL MEMBER USERMOD8.
*
* ADD THE FOLLOWING TO THE ROUTINES & ARGUMENTS TABLES.
*
* ROUTINES ENTRY      | ARGUMENTS(USRSLEEP) ENTRIES
* NAME = USRSLEEP     |
* LANG = ASSEMBLER    | 1) TIME          N    B    4
* FUNC = N            |
* TYPE =              |
* SYNT =              |
* LEN =               |
*
* * * * *
USRSLEEP DS          OD
          ENTRY USRSLEEP
          L          1,0(1)          ADDRESS ON PARM 1
          STIMER WAIT,BINTVL=(1)
          SR          15,15          SET RETURN CODE
          BR          14             RETURN TO CALLER
```

Sample Rule

```
SAMPSLEEP(TIME);
-  LOCAL Y;
-  -----
-  -----+-----
-  Y = $REALTIMER;                                | 1
-  CALL SLEEP(TIME);                              | 2
-  Y =($REALTIMER - Y) / 1000000;                  | 3
-  CALL ENDMSG('SLEPT FOR ' || $PIC(Y, 'NNN,NN9V.999') || | 4
-  ' SECONDS.');
```


Chapter 12

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, page 168](#)
- [Calling EMS, page 169](#)
- [Configuration, page 173](#)
- [Sample Applications, page 175](#)
- [Supported EMS Functions, page 177](#)

TIBCO Object Service Broker EMS Interface

Purpose of TIBCO Enterprise Message Service (EMS)

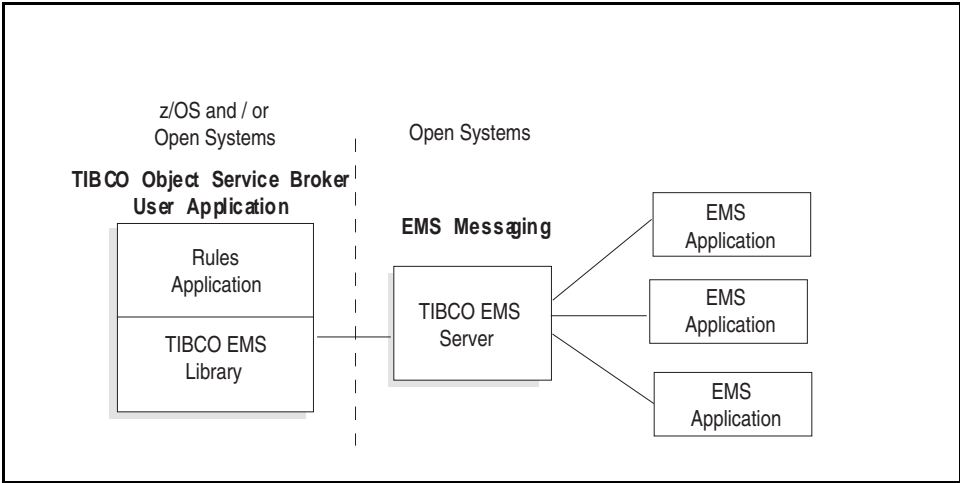
TIBCO Enterprise Message Service software lets application programs send and receive messages according to the Java Message Service (JMS) protocol. EMS is based on creation and delivery of messages. Messages are structured data that one application sends to another. The creator of the message is known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

Two shareable tools are provided to interface with the TIBCO EMS Client API. These tools are:

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
<code>tibems_byte;</code>	Binary of length 1
<code>tibems_short;</code>	Binary of length 2
<code>tibems_wchar;</code>	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

Some EMS routines require that the name of a data set or data set member containing encoded data to be used in SSL based message exchanges to be passed as an argument. For example:

```
tibems_status tibemsSSLParams_AddTrustedCertFile(  
    tibemsSSLParams SSLParams,  
    const char* filename,  
    tibems_int encoding );
```

In such cases, add a DD name statement declaration to the JCL that invokes the TIBCO Object Service Broker Execution Environment for each such data set or data set member required by the application, such as the following:

```
//SSLSCERT DD DISP=SHR,DSN=TIBCO.SXJ.V5R0M0.Z16.CNTL(SSLSCERT)  
//SSLCCERT DD DISP=SHR,DSN=TIBCO.SXJ.V5R0M0.Z16.CNTL(SSLCCERT)  
//SSLCKEY DD DISP=SHR,DSN=TIBCO.SXJ.V5R0M0.Z16.CNTL(SSLCKEY)
```

and pass a string such as the following:

```
DD:SSLSCERT
```

in which the 3 characters "DD:" are prepended to the DD name, as an argument to the EMS routines; in this case, as the filename argument to the routine tibemsSSLParams_AddTrustedCertFile.

Furthermore, EMS routines that accept such encoded data must have the encoding type explicitly stated, as automatic recognition of the encoding does not occur when using EMS under z/OS. This means that a value of 0 (zero, corresponding to the EMS C constant TIBEMS_SSL_ENCODING_AUTO) cannot be passed as the encoding argument above; one of the other numeric values corresponding to a certificate encoding must be used instead.

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 function 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 more information about the @MAP table and registering MAP areas.

Error Handling

Most EMS functions return a tibems_status code if EMS detects an error. The explanation of each of the status codes can be found in an appendix of *TIBCO Enterprise Message Service: C and COBOL Reference*.

If an abnormal termination occurs during rules processing TIBCO Object Service Broker creates an IBM LE formatted dump to report the problem. The dump appears as a SYSOUT file for the Execution Environment job. 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 exceptions.

Configuration

Initializing the EMS Interface

The Execution Environment establishes a special LE Enclave to run EMS calls and Execution Environment parameters exist to manage the environment. The TASKPOSIXNUM Execution Environment parameter determines if the enclave is started. By default the parameter is set to 0 and the LE Enclave is not started.

To enable EMS support for the Execution Environment set the TASKPOSIXNUM parameter to 1. The first call to EMS by a rule initializes the environment to run EMS and loads code related to invoking EMS.

Multi-threaded Environment

The LE Enclave used to run EMS calls is a multi-threaded USS environment. A thread is used to call EMS for any blocking call. Examples of such calls are `tibemsConnection_Create` and `tibemsMsgConsumer_Receive`.

Thread Processing

The THREADPOSIXNUM Execution Environment parameter defines the number of threads that will be started to handle blocking EMS calls. In effect it controls the number of concurrent EMS requests for an Execution Environment. If a thread is not available for an EMS call from a rule then the request is put on a wait queue until a thread is available. While a rule is waiting for an EMS call to complete, unless the session holds external resources, no Execution Environment interpreter tasks are blocked. The waiting session is rescheduled for execution when EMS completes the request.

Code Page Support

TIBCO Object Service Broker uses a single EBCDIC code page as defined in the @NLS1 table. Non-unicode text data is stored in this code page in the Data Object Broker table store. By default this code page is set to IBM-037. The host code page is automatically set to that specified in the @NLS1 table when EMS support in the Execution Environment is initialized.

The wire code page for EMS can be set using the EMSWIRECODEPAGE Execution Environment parameter (default is ISO8859-1). 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 TASKPOSIXNUM, THREADPOSIXNUM, and EMSWIRECODEPAGE Execution Environment parameters.

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 data set (member) 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 data set (member) 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 data set (member) 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_GetEMSMajorVersion		
tibemsConnectionMetaData_GetEMSPProviderName		
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_GetNPSSendCheckMode		
tibemsMsgProducer_GetPriority		
tibemsMsgProducer_GetTimeToLive		
tibemsMsgProducer_Send		
tibemsMsgProducer_SendEx		
tibemsMsgProducer_SendToDestination		
tibemsMsgProducer_SendToDestinationEx		
tibemsMsgProducer_SetDeliveryMode		
tibemsMsgProducer_SetDisableMessageID		
tibemsMsgProducer_SetDisableMessageTimestamp		
tibemsMsgProducer_SetNPSSendCheckMode		
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 13 **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, page 196](#)

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 z/OS Installing and Operating*.

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

Usage Notes

System Map Table

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

Required Local Variable

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

WebSphere MQ Environment

Only one MOM 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.

Possible TASKEXECNUM Increase

TIBCO Service Gateway for WMQ is TCB-specific. Therefore @MOMINIT sets TCB affinity for the transaction. This means an interpreter TCB is held for the duration of the transaction. In a multi-user Execution Environment, this can require you to increase the TASKEXECNUM Execution Environment parameter.

See Also *TIBCO Object Service Broker Parameters* for details on the TASKEXECNUM Execution Environment parameter.

Error Handling

The return code and reason code from the MOM software are stored in the map table @MQSMAP which is a shareable tool. You can check these stored codes in your rules. Refer to @MQSMAP in *TIBCO Object Service Broker Shareable Tools* for more information about this tool.

Example Rule

The following rule 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 14 **Introduction to the Call Level Interface**

This chapter describes the call level interface.

Topics

- [Aspects of the Call Level Interface, page 200](#)
- [Functionality of the Call Level Interface, page 203](#)
- [Operational Characteristics, page 206](#)
- [Call Level Interface Specification, page 207](#)
- [HRNHLLTM Module Parameters, page 209](#)
- [Valid Calling Sequences, page 212](#)
- [Examples of Typical Usage, page 214](#)
- [Using the Host Languages Interface, page 216](#)

Aspects of the Call Level Interface

Purpose of the Call Level Interface

The Call Level Interface provides facilities that you can code into client programs written in a third-generation language (3GL) to access a TIBCO Object Service Broker that is running in the same environment. These client programs can be running in batch, TSO, or CICS environments. Using these facilities, you can:

- Write TIBCO Object Service Broker user client routines in a 3GL such as assembler or COBOL
- Extend existing assembler or COBOL programs to include calls to the TIBCO Object Service Broker Call Level Interface

TIBCO Object Service Broker Software Development Kits (SDKs)

If your external program resides in a different address space from the TIBCO Object Service Broker it is accessing, you must use one of the TIBCO Object Service Broker SDKs. For more information:

External Environment	Interface	Refer to
C programs, C++ programs	SDK (C/C++)	Chapter 18, TIBCO Object Service Broker SDK (C/C++) Server, page 265. Chapter 19, Using TIBCO Object Service Broker SDK (C/C++), page 271.
Java programs	SDK (Java)	Chapter 20, Using TIBCO Object Service Broker SDK (Java), page 303.

Supported Functionality

With the Call Level Interface, you can:

- Start and stop an Execution Environment
- Start and stop a session within a started Execution Environment
- Start a transaction, modify its transactional characteristic and start additional transactions (streams) within the transaction
- Call a rule, commit or roll back table changes, and perform table access using the HLI or SQL preprocessor interface within a started transaction

All these functions provide feedback indicating the success of the operation in the form of a return code, a reason code, or a message.

Supported Connections

The Call Level Interface supports connections to the Execution Environment within the same address space (batch, TSO, or CICS environments). The interface is supported via standard z/OS calling conventions.

Shared Addressing

The user client routine can share an address space with the Execution Environment and the session. As a consequence, by using the MAP table interface:

- Rules are able to address common storage of the 3GL programs, if the addresses are passed to TIBCO Object Service Broker.
- Registered storage explicitly obtained by the rule as part of processing, or as data access triggered execution, are addressable by the 3GL program.

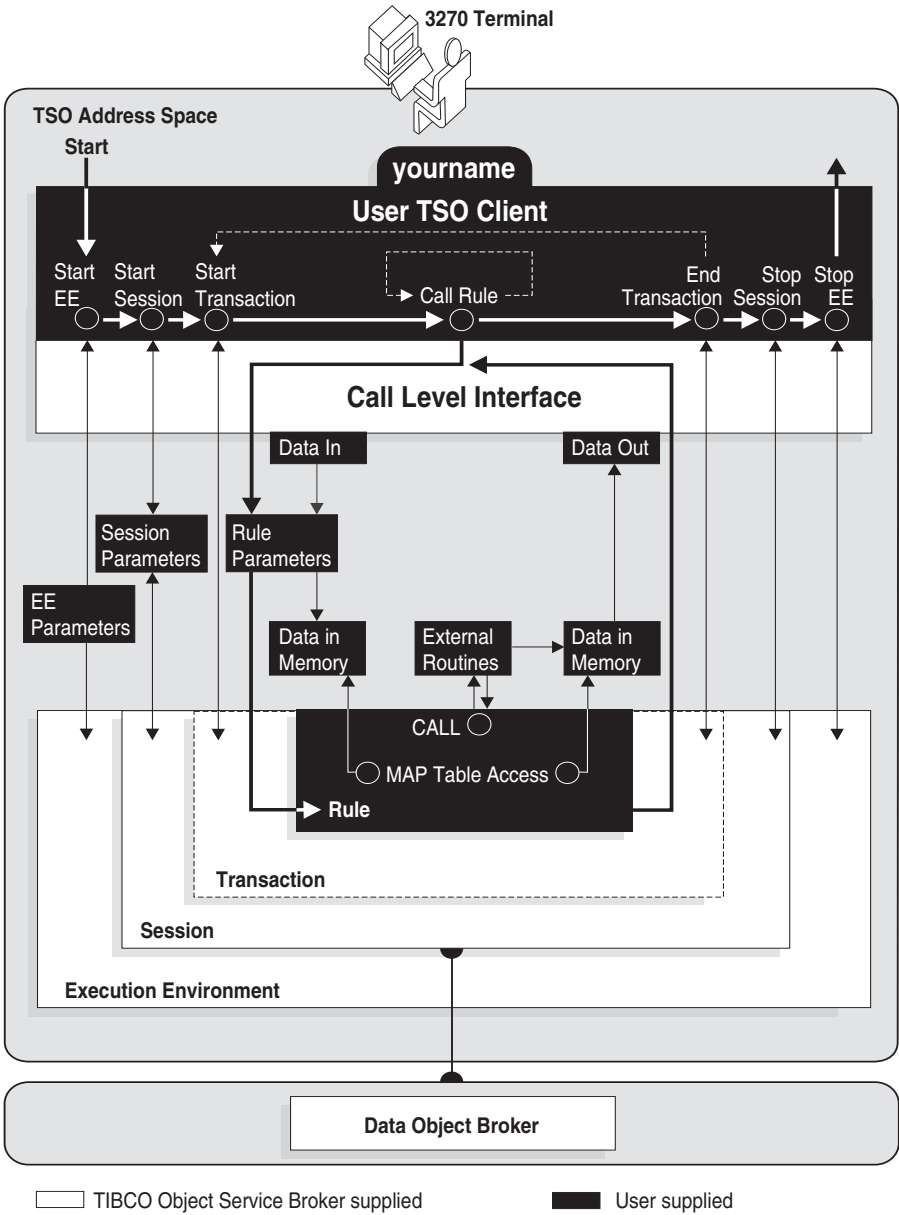
See Also *TIBCO Service Gateway for Files Installing and Operating* about MAP tables.
 TIBCO Object Service Broker Programming in Rules for information on rules.

Accessing Table Data Using the Host Languages Interface

If your client program is a COBOL program, you can embed TIBCO Object Service Broker access statements or SQL statements as outlined in [Chapter 18, TIBCO Object Service Broker SDK \(C/C++\) Server, page 265](#) to [Chapter 20, Using TIBCO Object Service Broker SDK \(Java\), page 303](#).

Illustration of Generic User Client Using Call Level Interface

The following illustration shows how a generic user client uses the Call Level Interface:



Functionality of the Call Level Interface

Start or Locate and Stop an Execution Environment

When starting an Execution Environment, you can pass it environment parameters. During the initialization, standby sessions are created that connect to the Data Object Broker and wait for requests to start a TIBCO Object Service Broker session.

Starting or Locating an Execution Environment

After the Execution Environment is started, TIBCO Object Service Broker returns to your client program the name of the Table Data Store (TDS) to which your Execution Environment is connected. When locating an Execution Environment, a handle to an Execution Environment that is already started is returned to the client program.

Stopping an Execution Environment

When the Execution Environment is stopped, all sessions are terminated and disconnected from the Data Object Broker. Your handle to the Execution Environment is no longer valid.

Start and Stop a TIBCO Object Service Broker Session

To start a session, you need a valid handle to an Execution Environment. When starting a session you can pass it session parameters. During session initialization, an available standby session is acquired, login security and profile processing is performed, and the login transaction is completed. After the session is started, the handle to the session is released back to the client program.

When stopping the session, the session is released back to the pool of standby sessions and control is returned back to the client program. Your handle to the session is no longer valid.

Start and End a TIBCO Object Service Broker Stream

A stream is a transaction nesting level within which you can modify its transactional characteristics, and start and end a transaction. When you start a stream, the transaction nesting level increases by one and it temporarily suspends the ability to access the parent transaction context. When you end a stream, it decreases the nesting level by one and resets the transaction context to that of the previous stream level.

Modifying Stream Characteristics

When starting a stream, you can specify the following characteristics:

- BROWSE or UPDATE
- TEST or NOTEST
- Local LIBRARY
- SEARCH order of the rules libraries

These characteristics are inherited from the user profile and the parent stream in the same way that the EXECUTE statement provides for inheritance. You can modify stream characteristics when the stream does not contain a started transaction.

Start and End a TIBCO Object Service Broker Transaction

The first transaction starts a transaction stream. You can nest transactions within transactions. Each nesting causes a transaction stream to start.

When you start a transaction stream the transaction nesting level increases by one and it temporarily suspends the ability to access the parent transaction context. When you end a stream, it decreases the nesting level by one, and resets the transaction context to that of the previous stream level.

Modifying Transaction Characteristics

When starting a transaction, you can specify the following characteristics:

- BROWSE or UPDATE
- TEST or NOTEST
- Local LIBRARY
- SEARCH order of the rules libraries

These characteristics are inherited from the user profile and the parent stream in the same way that the EXECUTE statement provides for inheritance. When a transaction ends, changes made to persistent table data are committed and all locks are dropped.

Committing or Rolling Back Persistent Table Changes

You can commit persistent table changes to TDS and external database tables; logical locks acquired during the course of the transaction are retained after the commit. You can also roll back and discard all changes to TDS and external database tables made since the last commit or transaction start.

Call a TIBCO Object Service Broker Rule

You specify the name of an entry rule to be called and its arguments. When the rule completes within the transaction environment, control is returned back to the client program. In addition, the client program can pass DATA-IN and DATA-OUT areas to the session. The rule can read the DATA-IN areas using MAP tables or tools such as [\\$GETENVCOMMAREA](#). The rule can write to DATA-OUT areas using MAP tables or tools such as [\\$SETENVCOMMAREA](#). The rule can also access other data in the address space if the storage is registered to the MAP table facilities.

Finding the Name of a Rule in A Transaction

You can use the [\\$RULENAME](#) shareable tool to find out the name of a rule in the current transaction or in a parent transaction.

See Also *TIBCO Object Service Broker Managing Data* about MAP tables.
 TIBCO Object Service Broker Shareable Tools about the tools.

Operational Characteristics

Supported 3GL Languages

The user client program can be written in any language. To also access data using the Host Languages Interface, the program is restricted to COBOL.

Multiple Execution Environments per Address Space

Connections to an Execution Environment are restricted to one Execution Environment for that address space. However, multiple-session connects are allowed.

Standby Sessions

In multiple-session environments such as CICS, the number of programs allowed to issue TIBCO Object Service Broker session interface calls is determined by the STANDBYNUM Execution Environment parameter, and is limited to the number of standby sessions started at Execution Environment startup. If there are more calling programs than the number of available standby sessions, the affected calling programs are put in a wait queue until one is available.

When Viewed by TIBCO Object Service Broker Administrator Tools

Standby sessions that are connected to the Data Object Broker and are not in use by a client can be identified by the format of their session name:

\$aaaaannn

\$	Is fixed and identifies that this is a standby session
aaaa	Is the fixed address space ID (ASID) of the Execution Environment
nnn	Is a generated sequence number

A standby session assumes the user profile specified in the STARTSS. Therefore, multiple standby sessions can assume similar profiles from the same user ID.

A standby session remains connected until the calling program that started the session ends it. The administrator tools see only the registered user ID, starting with a dollar sign (\$).

Call Level Interface Specification

What Is the Module to Call?

You access all facilities of the Call Level Interface by calling the module HRNHLLTM. HRNHLLTM requires eight standard parameters, described in [HRNHLLTM Module Parameters on page 209](#). Additional parameters are optional, and are used to convey exit information for advanced applications (refer to [Chapter 17, Multiple-Session Execution Environments in Batch, on page 255](#)).

Standard operating system parameter passing and linkage conventions are used. HRNHLLTM is link-edited with the calling application program. Refer to the sample members COBCOLNK in the JCL data set distributed with TIBCO Object Service Broker for an example link-edit of a user client program with the Call Level Interface modules.

Example CALL Formats

COBOL Example

```
CALL 'HRNHLLTM'      USING HRNHLLWA HRN-HRN-OPERATION HRN-OPERAND
HRN-PARM HRN-DATA-IN HRN-DATA-OUT HRN-RETURN-DATA HRN-RETURN-CODE.
```

Assembler Example

```
CALL HRNHLLTM, (HRNHLLWA, OPERATION, OPERAND, PARM, DATA-IN,
DATA-OUT, RETURN-DATA, RETURN-CODE), VL, MF=(E, CALLLIST)
```

Required Parameters

Every call to HRNHLLTM must provide a valid:

- HRNHLLWA work area
- OPERATION
- RETURN-DATA and RETURN-CODE

Depending on the Call Level Interface function, the other parameters are ignored.

Usage of the Parameters

The call parameters passed into HRNHLLTM are used to:

- Convey the TIBCO Object Service Broker context of the client program (HRNHLLWA)
- Specify the Call Level Interface function (OPERATION and OPERAND)
- Communicate data between the client and TIBCO Object Service Broker (PARM, DATA-IN, DATA-OUT, RETURN-DATA, and RETURN-CODE)

Parameter Types

Parameters are of type IN, OUT, or IN/OUT. Storage for all parameters must be allocated by the client program.

- IN-type parameters are set by the client and read by the Call Level Interface.
- OUT-type parameters are set by the Call Level Interface and read by the client.
- IN/OUT-type parameters are set and read by the client and by the Call Level Interface.

NULL Parameters

Some Call Level Interface functions require a NULL call parameter. In this case, a dummy value must be coded in that position. The value of the dummy parameter is ignored.

HRNHLLTM Module Parameters

Valid Call Parameters

The following table provides a description of each parameter to HRNHLLTM and tells whether each parameter is of type IN, OUT, or IN/OUT.

Parameter	Description	Type
HRNHLLWA	Work area of 687-bytes, used by the Call Level Interface to maintain the handles to the Execution Environment, session, stream and transaction and cursor context. This work area is in the same format as that used by the Host Languages Interface.	IN/OUT
OPERATION	Uppercase character string naming the type of interface call to be performed. For example, the string STARTEE designates the Start Execution Environment operation; and STOPEE, the Stop Execution Environment operation. Refer to Operational Parameters on page 210 for a listing of call OPERATION parameters.	IN
OPERAND	Uppercase character string used as an operand of the OPERATION to further qualify the operation. For example, the string BATCH qualifies the STARTEE operation.	IN
PARM	Variable length uppercase character string used as a parameter to the OPERATION/OPERAND. For example, the assembler string: <code>AL2(10),C'TDS=DEV3'</code> can be used as a PARM to the STARTEE BATCH function to specify that the Execution Environment is to connect to a Data Object Broker called DEV3. The variable-length string is made up of a two-byte header containing the length of the string and the string that must immediately follow the header. An empty string of length zero is represented by two-byte header with a value of binary 0. Rules arguments enclosed in quotes can contain lowercase data.	IN

Parameter	Description	Type
DATA-IN	This parameter is used only when calling a rule. In this case, DATA-IN is used to pass blocks of data from the client to the session. Refer to Calling a Rule – CALLRULE on page 249 for a detailed description of this parameter.	IN
DATA-OUT	This parameter is used only when calling a rule. In this case DATA-OUT is used to pass data from the session to the client. Refer to Calling a Rule – CALLRULE on page 249 for a detailed description of this parameter.	OUT
RETURN-DATA	For some interface functions, if the RETURN-CODE is 0, this parameter contains feedback data. If the RETURN-CODE is non-zero, the parameter contains either a reason-code or a variable length message. RETURN-DATA should be at least 159 bytes long to accommodate the largest possible return message.	OUT
RETURN-CODE	Four-byte return code returned by the interface indicating the success of the requested operation. A value of 0 indicates success; any other value indicates that the operation did not complete successfully, with the exception of the STARTEE and STOPEE calls. If the return code is non-zero, the RETURN-DATA parameter can contain a reason code or a message. Refer to Starting or Locating the Execution Environment – STARTEE on page 234 and Stopping the Execution Environment – STOPEE on page 237 for further explanation of return codes in the STARTEE and STOPEE calls.	OUT

Operational Parameters

The following table relates each Call Level Interface function to an OPERATION and OPERAND parameter value. Ignored parameters are blank in the table.

Interface Function	Operation	Operand	Parameter	DATA-IN	DATA-OUT	DATA-RETURN (if return code = 0)
Start or locate an Execution Environment	STARTEE	BATCH TSO CICS	Execution Environment parameters			TDS name
Start a session	STARTSS		Session parameters			User ID
Start a transaction /stream and set its characteristics	STARTTR		{BROWSE UPDATE} {TEST NOTEST} SEARCH=p LIBRARY=xxxxxxx			
Start a rule	CALLRULE		Rule and its arguments	Block of data	Block of data	
End transaction/stream	STOPTR	COMMIT ROLLBACK				
Stop a session	STOPSS					
Stop an Execution Environment	STOPEE					

Valid Calling Sequences

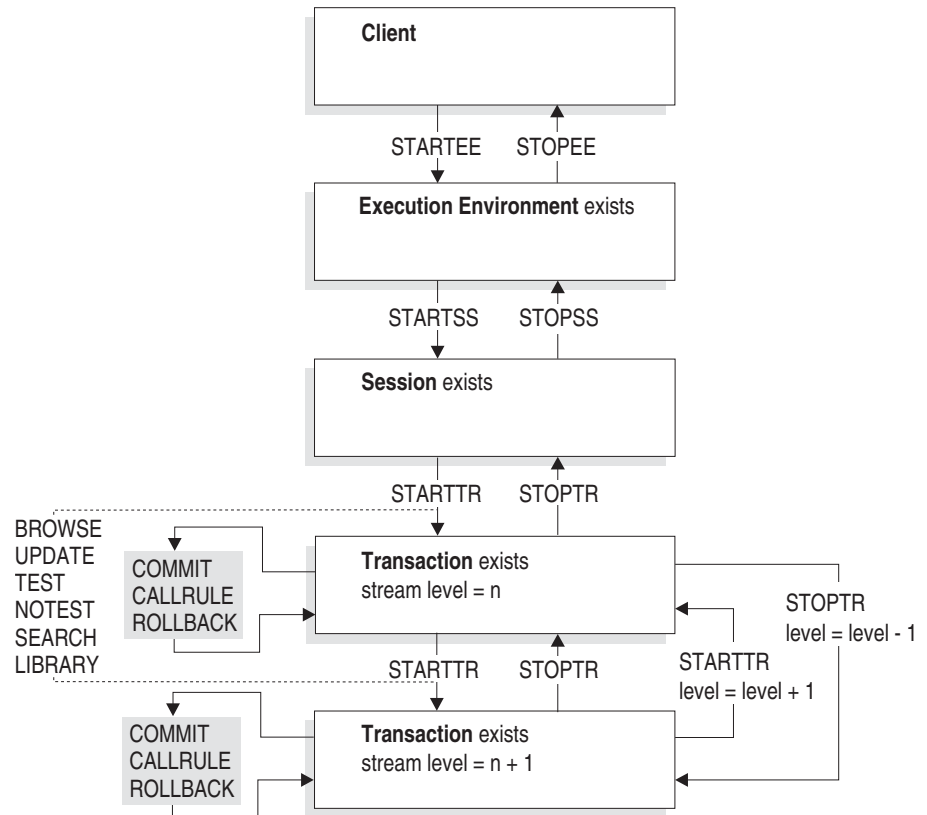
You must call Call Level Interface functions in a particular sequence. For example, before you can start a session, you must first start an Execution Environment. In general, you cannot use the functionality of a TIBCO Object Service Broker object until it has been explicitly created by starting it and you cannot create a TIBCO Object Service Broker object until the appropriate environment has been established.

Calling Sequence

External Environment	Calling Sequence
Single-session (Batch and TSO)	<p>The client program must:</p> <p>Establish the Execution Environment.</p> <p>Start the session.</p> <p>Invoke the TIBCO Object Service Broker functionality.</p> <p>Termination of sessions and of the Execution Environment is performed in the reverse sequence.</p>
Multiple-session (CICS)	<p>In a multiple-session environment, the Execution Environment must be started separately, which is normally done as part of standard operations. This process establishes, at start-up, the concurrent number of standby sessions deemed necessary for the connect of the interface.</p> <p>The client program, using the STARTEE function:</p> <p>Locates the Execution Environment.</p> <p>Starts a session (which acquires one of the available standby sessions).</p> <p>Invokes the TIBCO Object Service Broker functionality.</p> <p>The STOPSS function terminates the session. Session termination releases the standby session and makes it available to other calling programs.</p>

Permissible Transitions Between the Call Level Interface Functions

The following illustration shows the permissible transitions between the Call Level Interface functions. Invalid calling sequences generate a non-zero RETURN-CODE.



Examples of Typical Usage

You can use the Call Level Interface to write a client program that provides features of the batch client and the rules language, but with a much finer line of granularity. For economy of representation, only the OPERATION, OPERAND, and PARM (if required) are explicitly stated. The COBOL or assembler call, null parameters and RETURN-DATA and RETURN-MESSAGE processing should be viewed as implicit.

Batch Client Example

In this example, the functionality of a simple invocation of the z/OS batch client is translated into a series of Call Level Interface calls. Assume that we invoked the batch client as follows (the session parameter NOBROWSE means UPDATE):

```
// EXEC  PGM=S6BBATCH,
//      PARM='TDS=DOB1,RULE=EMP_ADD(2),LIBRARY=EMPTEST,NOBROWSE,SEARCH=L,NOTEST'
```

This functionality could be implemented by a client program as follows:

```
STARTEE      'TDS=DOB1'
STARTSS
STARTTR UPDATE, NOTEST, LIBRARY=EMPTEST, SEARCH=L
CALLRULE     EMP_ADD(2)
STOPTR
STOPSS
STOPEE
```

Nested Execute Example

This example shows how to create a nested transaction. Based on the [Batch Client Example on page 214](#), assume that we invoke a BROWSE transaction and call the EMP_B rule before terminating the initial transaction. This functionality could be implemented by a client program as follows:

```

STARTEE          'TDS=DOB1 '
STARTSS
STARTTR UPDATE, NOTEST, LIBRARY=EMPTTEST, SEARCH=L
CALLRULE         EMP_ADD(2)
.
.
STARTTR BROWSE, NOTEST, LIBRARY=EMPTTEST, SEARCH=L
CALLRULE EMP_B
STOPTR
STOPTR
STOPSS
STOPEE

```

TRANSFERCALL Example

This example shows how to obtain the rules-based functionality of a TRANSFERCALL using the Call Level Interface. Based on the [Batch Client Example on page 214](#), assume that we invoke a BROWSE transaction and call the EMP_B rule after terminating the initial transaction. This functionality could be implemented by a client program as follows:

```

STARTEE          'TDS=DOB1 '
STARTSS
STARTTR UPDATE, NOTEST, LIBRARY=EMPTTEST, SEARCH=L
CALLRULE         EMP_ADD(2)
STOPTR
STARTTR BROWSE
CALLRULE         EMP_B
STOPTR
STOPSS
STOPEE

```

Using the Host Languages Interface

User COBOL source programs can use the Call Level Interface together with either embedded TIBCO Object Service Broker access statements or embedded SQL statements. After establishing a transaction environment using the Call Level Interface STARTTR function, you can issue embedded TIBCO Object Service Broker or SQL statements to access TIBCO Object Service Broker tables.

Writing a COBOL Program Using a Combination of the Call Level Interface, TIBCO Object Service Broker Access Statements, and SQL Statements

The procedure for writing a COBOL program that uses the Call Level Interface and TIBCO Object Service Broker access or embedded SQL statements is:

1. In the WORKING-STORAGE section, establish the access environment for embedded access statements.

Refer to [Chapter 21, Coding TIBCO Object Service Broker Access Statements, on page 339](#) or [Chapter 22, Coding SQL Access Statements, on page 351](#), depending on whether you intend to use TIBCO Object Service Broker access statements or SQL access statements. In either case, the HRNHLLWA, which is shared between the Call Level Interface and the access statements, is automatically generated for you.

2. In the WORKING-STORAGE section, copy in the copybook HRNHLLIST provided in the distribution MACRO data set.

This copybook provides you with the additional declarations used by the Call Level Interface.

3. In the PROCEDURE DIVISION, create a transactional environment as described in the previous sections before coding data access statements.

You are in a transactional environment after issuing the STARTTR function.

4. In the transactional environment, code access statements.

Refer to [Chapter 21, Coding TIBCO Object Service Broker Access Statements, on page 339](#) or [Chapter 22, Coding SQL Access Statements, on page 351](#).

During program execution, these access statements can be preceded or followed by the Call Level Interface CALLRULE function.

When you end a first-stream-level transaction using the STOPTR function, you can no longer execute TIBCO Object Service Broker access statements or SQL access statements. You can, however, resume coding these statements after a STOPTR function, if this returns you to a lower transaction nesting level.

Additional Steps

1. Precompile using the preprocessor

Precompile third-party programs with embedded data access calls through the TIBCO Object Service Broker preprocessor, to allow for the TIBCO Object Service Broker or SQL data access statements.

Convert the TIBCO Object Service Broker or SQL statements to COBOL using the [HLIPREPROCESSOR](#) tool. Refer to [Chapter 23, Processing COBOL Programs, on page 367](#) for information on how to use this tool.

2. Compile and link your program

Refer to the member COBCOLNK in the JCL data set distributed with TIBCO Object Service Broker, for an example of this step.

3. Invoke the COBOL user client program

Refer to the member COBBATCH in the JCL data set distributed with TIBCO Object Service Broker, for an example of this invocation.



The STEPLIB DD must have the TIBCO Object Service Broker Load Library concatenated to it. Since such a concatenation normally causes de-authorization of the STEPLIB, you should also provide a HRNLIB DD pointing to the TIBCO Object Service Broker Load Library.

Chapter 15 **Preparing the Environment, Analyzing Returned Values, and Modifying Changes**

This chapter describes how to prepare the environment, analyze returned values, and modify changes.

Topics

- [Preparing to Start or Locate the Execution Environment, page 220](#)
- [How to Analyze the Return and Reason Codes, and Returned Message, page 222](#)
- [Call Level Interface Return Codes, page 225](#)
- [Call Level Interface Reason Codes, page 226](#)
- [Committing and Rolling Back Persistent Table Changes, page 231](#)

Preparing to Start or Locate the Execution Environment

Preparatory Steps

Complete the following tasks, starting with [Allocate and Initialize Storage on page 221](#), before you start or locate an Execution Environment in your client program using the STARTEE function. Sample code is included to assist you with developing your applications for the Call Level Interface. This code appears in the MACRO and COBOL data sets.

MACRO Data Set

Member Name	Description
HRNHLLIST	Copy book that defines constructs and parameter names for COBOL clients calling HRNLLTM.
HRNHLWAA	Copy book that defines an initialized HRNHLLWA for assembler clients.
HRNHLWAC	Copy book that allocates and initializes the HRNHLLWA work area in COBOL before it is used by the HRNHLLTM program.

COBOL Data Set

Member Name	Description
COBCAPI1	A program written using the current Call Level Interface statements to start, prepare, and stop an Execution Environment. It also checks and responds to the return and reason codes issued by these operations.
COBBATCH	Batch JCL for the Call Level Interface.
COBCOLNK	JCL to pretranslate, compile, and link the sample COBOL program COBCAPI1.

Member Name	Description
S6BCALIN	A Call Level Interface assembler program for the CICS Execution Environment.
S6BCAPID	Driver for testing the Call Level Interface in batch or from TSO (assembler version).

Task A Allocate and Initialize Storage

You must allocate and initialize storage for the work area HRNHLLWA, to contain handles to all the TIBCO Object Service Broker objects.

Task B Allocate DD Names

If you are starting an Execution Environment using the TSO or BATCH operand, you must ensure that all the DD names required by the Execution Environment are allocated either through JCL or dynamic allocation:

1. Ensure that the TIBCO Object Service Broker load library is concatenated to the STEPLIB.

If the other libraries in the STEPLIB are not authorized, add a HRNLIB DD statement to point to the load library to run an authorized Execution Environment.

2. Ensure that the DDname HRNIN points to an Execution Environment parameter file.

Task C CICS only – Establish the Execution Environment

If your client program is intended for execution in the CICS environment, the Execution Environment must be established at CICS initialization time, or via the HINT transaction. Refer to [Chapter 7, Using the TIBCO Service Gateway for CICS, on page 59](#) for information on establishing a CICS Execution Environment.

How to Analyze the Return and Reason Codes, and Returned Message

After every call to HRNHLLTM you should check the return code to ensure that the operation was successful. The Call Level Interface return code is set in the eighth parameter to HRNHLLTM (HRN-RETURN-CODE in the COBOL examples, CODE in the assembler examples). Possible values of return codes and reason codes and their meanings are listed in [Call Level Interface Return Codes on page 225](#) and in [Call Level Interface Reason Codes on page 226](#).

Evaluation of the Return and Reason Codes

The return and reason codes are evaluated as follows:

- If the return code is not zero, either a numeric reason code is returned, or a message string is returned in the seventh parameter (S6BRETURN-DATA in the COBOL examples, DATA in the assembler examples).
- If the first byte of RETURN-DATA is zero, the first four bytes of RETURN-DATA contain the reason code.
- If the first byte of the reason code is not zero, a printable message string of length 129 is returned. This message is the ENDMSG—a short error message that typically appears at the bottom of the workbench during an online TIBCO Object Service Broker session.

Capturing the Returned Values

Alternatively, you can use the [MESSAGE_LOG](#) tool to capture a message log and place it in a TDS table for later viewing. This is an appropriate technique during application development and debugging. You can also capture relevant parts of the message log and place it in a commarea to pass back to the COBOL program. This requires the ability to predict the error and handle it in your client program, but it is appropriate for error handling in a production system.

Examples

The following two examples show how you can analyze and display the return code, reason code, and returned message.

Analyzing and Displaying in COBOL

```

OSB-STATUS.
  DISPLAY 'CHECKING OSB STATUS'.
  IF OSB-CALL-OK
    NEXT SENTENCE
  ELSE
    IF REASON-CODE-EXISTS
      DISPLAY '*****'
      DISPLAY '***      Unsuccessful API Call      ***'
      DISPLAY '*****'
      DISPLAY 'HRN-RETURN-CODE = ' HRN-RETURN-CODE
      DISPLAY 'HRN-REASON-CODE = ' HRN-REASON-CODE
      STOP RUN
    ELSE
      DISPLAY '*****'
      DISPLAY '***      Unsuccessful OSB Call      ***'
      DISPLAY '*****'
      DISPLAY 'HRN-RETURN-CODE = ' HRN-RETURN-CODE
      DISPLAY 'HRN-RETURN-DATA = ' HRN-RETURN-DATA
      STOP RUN.
OSB-STATUS-EXIT.
EXIT.

```

Analyzing and Displaying in Assembler

```

* - - - - - *
*
*   SUBROUTINE - WRITE OUTPUT
*
* - - - - - *

S2SHOWRC DS      0H
          ST      R14,STRLINK2      SAVE LINK REGISTER
*
          MVC     STROUREC,BLANKS    BLANK OUTPUT AREA
          MVC     STRETURN,RETURNCO  COPY 'RETURN CODE='
          L       R1,HAPRCODE        GET RETURN CODE ADDRESS
          HEXCHAR  2(R1),STRETCOD,2
*
          L       R1,HAPRDATA        GET REASON CODE ADDRESS
          CLI     0(R1),X'00'        REASON CODE?
          BNE     S2SHPUT1           NO, JUST PRINT THE RETURN CODE
*
          MVC     STREASON,REASONCO  COPY 'REASON CODE='
          HEXCHAR  2(R1),STREACOD,2
*
S2SHPUT1 DS      0H
          ADRMODE 24                  USE AMODE(24) FOR PUT REQUEST
          PUT     (OUTREG),STROUREC
          ADRMODE 31                  RESTORE AMODE(31)
*

```

```

L      R1,HAPRDATA      GET REASON CODE ADDRESS
CLI    0(R1),X'00'      WAS IT REASON CODE?
BE     S2SDATAO         YES, DONE
*
MVC    STRRDATA,RETURND COPY 'RETURN DATA='
MVC    STRETDAT,0(R1)   COPY RETURN DATA
MVC    RETDATA(80),BLANKS BLANK OUTPUT AREA
*
ADRMODE 24              USE AMODE(24) FOR PUT REQUEST
PUT     (OUTREG),STROUREC
ADRMODE 31              RESTORE AMODE(31)
*
S2SDATAO DS      0H
```

Call Level Interface Return Codes

Listing and Explanation

The following table lists the return codes. A listing of symbols is available for your use in member HRNAPIRC of the MACRO data set.

Return Code	Explanation
00	Request successful.
01	No operation code specified.
02	Invalid operation request.
03	STARTEE request failed.
04	STARTSS request failed: reason code in the RETURN-DATA area.
05	DO request failed: reason code in the RETURN-DATA area.
06	Execution Environment not started, but located.
07	STOPEE request failed.
08	HLLTAM routine not installed (User error) (returned when using the TIBCO Object Service Broker Host Language Interface).
09	STOPSS request failed (reason code in the RETURN-DATA area).
10	STARTTR request failed.
11	CALLRULE request failed.
12	STOPTR request failed.
13	Session is canceled or terminated.

Call Level Interface Reason Codes

Listing and Explanation

The following tables list the reason codes. A listing of symbols is available for your use in member HRNAPIRC of the MACRO data set.

Common Reason

Reason Code	Explanation
32	Invalid Execution Environment type.
33	Execution Environment is not active.
34	Interface session is not active.
35	Interface calls not allowed from within the Execution Environment.
36	Interface calls are out of sequence.
37	No standby sessions active in the Execution Environment.
38	Input commarea storage is inaccessible.
39	Output commarea storage is inaccessible.
40	Input commarea length error.
41	Output commarea length error.
42	STOPEE already in progress.

StartEE Fail Reasons

Reason Code	Explanation
48	No Execution Environment specified.
49	TIBCO Object Service Broker CICS interface is not active.

Reason Code	Explanation
50	TIBCO Object Service Broker internal error.
51	Execution Environment initialization failed.
52	Locate CICS callers EIB failed.

Start Session Fail Reasons

Reason Code	Explanation
64	Execution Environment is in quiesce mode.
65	Invalid session parameters.
66	TIBCO Object Service Broker user ID is longer than eight characters.
67	Invalid character set name.
68	Invalid execution mode.
69	Security login fail.
70	Security logout fail.
71	Session scope storage initialization fail.
72	Session scope storage termination fail.
73	Session rejected by user exit.
74	Unknown return code from user exit.

DO Fail Reasons

Reason Code	Explanation
80	Invalid interface DO function.
81	Invalid DO parameters.
82	STARTTR not issued.
83	STARTTR failed, transaction already active.
84	DO or transaction option not specified.
85	DO or transaction option not supplied.
86	Invalid transaction option.
87	Transaction options not accepted within a transaction.
88	Required transaction option parameter not supplied.
89	Invalid transaction option keyword parameter length.
96	Rule name not supplied.
97	Rule name length error.
98	Invalid stop transaction parameter length.
99	Invalid stop transaction parameter.
100	No stream work area (SMS) exists.
101	Table access method not active.
102	Commit failed.
103	Rollback failed.
104	Rules argument syntax error.
105	Host Languages Interface runtime environment does not exist (that is, no CALLRULE was issued).
106	Attempt to nest too many transactions

Stop Session Fail Reasons

Reason Code	Explanation
128	Stream and/or transaction still active.

StopEE Fail Reasons

Reason Code	Explanation
144	Execution Environment not stopped, but the STARTEE count is decremented.

Call Level Interface Client/Server fail Reason Codes

Reason Code	Explanation
160	Incompatible Call Level Interface Client/Server version.
161	Unsupported code page specified.
162	Invalid Endian type specified.
163	Session control storage not available.
164	Start CICS task failed.
165	Environment initialization error.
166	Call Level Interface message length exceeds maximum.
167	Call Level Interface message storage not available.
168	DATAOUT length exceeds maximum.
169	DATAOUT storage not available.
170	User ID not supplied.

Reason Code	Explanation
171	User ID or password is not valid. See Appendix A, SDK (C/C++) and SDK (Java) Error Reason Codes, on page 373 for details.
172	User ID suspended.
173	User ID cannot access TIBCO Object Service Broker at this time.
174	Password not supplied.
176	Password expired, new password missing.
177	New password not valid.
178	Password upgrade fail.
179	Password decrypt fail.

Committing and Rolling Back Persistent Table Changes

If you specify COMMIT or ROLLBACK as the parameter to the STOPTR operation, your client program can issue a COMMIT or ROLLBACK.

Notes

Within a started transaction, and before it is ended, you can COMMIT or ROLLBACK changes made by the data access statements INSERT, REPLACE, and DELETE to persistent TDS or external database tables.

Sample Calls

COBOL Call to Perform a COMMIT

```
COPY HRNHLWAC
COPY HRNHLLIST
.
.
01 HRN-STOPPRARM.
   05 STOPPARMLEN                PIC9(4) USAGE COMP-4 VALUE 6.
   05 STOPPARMDAT                PICX(6) VALUE 'COMMIT'.
.
.
CALL 'HRNHLLTM' USING HRNHLLWA 'STOPTR'
   HRN-NULL-LIST HRN-STOPPARM HRN-NULL-LIST HRN-NULL-LIST
   HRN-RETURN-DATA HRN-RETURN-CODE.
```

Non Re-entrant Assembler Call to Perform a COMMIT

```
CALL HRNHLLTM, (HRNHLLWA, C'STOPTR', 0, STOPPARM, 0, 0, DATA, CODE), VL
.
.
STOPPARM CD AL2(6), C'COMMIT'
DATA DS CL180 REASON CODE FROM HRNHLLTM
CODE DS F RETURN CODE FROM HRNHLLTM
```

Returned Values

Return Codes	Description
0	STOPTR with COMMIT or ROLLBACK request succeeded.
9	STOPTR with COMMIT or ROLLBACK request failed. The first four bytes of the RETURN-DATA parameter are set to the reason code. Refer to Call Level Interface Reason Codes on page 226 .

Chapter 16 **Call Level Interface Functions**

This chapter describes the call level interface functions.

Topics

- [Starting or Locating the Execution Environment – STARTEE, page 234](#)
- [Stopping the Execution Environment – STOPEE, page 237](#)
- [Starting the Session – STARTSS, page 239](#)
- [Stopping the Session – STOPSS, page 241](#)
- [Starting a Transaction – STARTTR, page 243](#)
- [Modifying Transactional Characteristics, page 245](#)
- [Ending a Transaction – STOPTR, page 247](#)
- [Calling a Rule – CALLRULE, page 249](#)

Starting or Locating the Execution Environment – STARTEE

Syntax

For the syntax of all Call Level Interface functions, see [Call Level Interface Specification on page 207](#) and [HRNHLLTM Module Parameters on page 209](#).

Calling Parameters

After preparing to start the Execution Environment, your client program starts or locates the Execution Environment by specifying STARTEE as the OPERATION.



- For TSO, batch, and CICS, if an Execution Environment is already active in the address space, the STARTEE operation locates the Execution Environment and copies a handle to it into the HRNHLLWA workarea.
- For TSO and batch Execution Environments, if no Execution Environment exists, a same-address space Execution Environment is started.

Starting an Execution Environment

You start either a TSO or batch Execution Environment by specifying TSO or BATCH as the OPERAND parameter. Overrides to the Execution Environment parameters are supplied through the variable length PARM parameter.



The Call Level Interface does not support starting a CICS Execution Environment.

Obtaining Execution Environment Startup Parameters

When starting an Execution Environment, the PARM parameter contains the Execution Environment startup parameter. This information is obtained according to the following order of precedence:

1. EECONFIG member of the JCL data set
2. HRNIN data set
3. Parameter string, for example, *TDS=HCDL1000*

On successful execution, the value of RETURN-CODE is 0, and the first eight characters of RETURN-DATA are the name of the Table Data Store (TDS) to which your Execution Environment is connected, 'HCDL1000' in our example.

See Also *TIBCO Object Service Broker Parameters* about Execution Environment parameters.

Locating an Execution Environment

You locate an Execution Environment by specifying its type as the OPERAND (BATCH, TSO, or CICS). Overrides to the Execution Environment parameters supplied through the variable length PARM are ignored when locating an Execution Environment.

Sample COBOL Call

```

      .
COPY  HRNHLWAC.
COPY  HRNHLLIST.
      .
01    HRN-RGNPARM.
      05  RGNPARMLEN          PIC 9(4)  USAGE COMP-4  VALUE 12.
      05  RGNPARMDAT          PIC X(12)  VALUE 'TDS=HCDL1000'.
      .
      .
      .
CALL  'HRNHLLTM'  USING HRNHLLWA 'STARTEE' 'BATCH'
      HRN-RGNPARM  HRN-NULL-LIST HRN-NULL-LIST
      HRN-RETURN-DATA HRN-RETURN-CODE.
      .

```

Sample Non Re-entrant Assembler Call

```
CALL  HRNHLLTM, (HRNHLLWA, C 'STARTEE ' , C ' BATCH ' ,
                PARM, 0, 0, DATA, CODE) , VL
.
.
.
PARM    DC  AL2(12), C 'TDS=HCDL1000'
DATA    DS  CL159          REASON CODE FROM HRNHLLTM
CODE    DS  F              RETURN CODE FROM HRNHLLTM
.
```

Returned Values

Return Code	Description
0	STARTEE request succeeded. The first eight bytes of RETURN-DATA are set to the name of the TDS to which your Execution Environment is connected (in the example it would be set to HCDL1000).
3	STARTEE request failed. The first four bytes of the RETURN-DATA parameter are set to the reason code. Refer to Call Level Interface Reason Codes on page 226 .
6	An existing Execution Environment was located. Note The first eight bytes of the RETURN-DATA parameter could be different from what you specified in the TDS parameter—they contain the name of the Data Object Broker to which the active Execution Environment is currently connected.

Advanced STARTEE Batch Usage

To interface with certain kinds of third-party environments, you can create a multiple-session non-blocking batch Execution Environment by specifying an environmental wait routine specific to the third-party environment. Using the assembler interface, you can pass the address of the environmental wait routine to the Execution Environment by specifying an exit descriptor as the ninth parameter to HRNHLLTM. The exit descriptor is mapped by the assembler DSECT called HRNXD and distributed in the MACRO distribution data set. It is of Exit type SESENVWT. In addition, you can pass in an environmental anchor block that can be subsequently accessed by other sessions executing in the Execution Environment.

Additional Information

For an example of setting up an environmental wait routine, refer to the member S6BCAPID, in the ASM data set distributed with TIBCO Object Service Broker. For a complete description of this facility, refer to [Chapter 17, Multiple-Session Execution Environments in Batch, on page 255](#).

Stopping the Execution Environment – STOPEE

Syntax

For the syntax of all Call Level Interface functions, see [Call Level Interface Specification on page 207](#) and [HRNHLLTM Module Parameters on page 209](#).

Calling Parameters

When all Call Level Interface sessions are terminated in the Execution Environment, your client program stops the Execution Environment by specifying STOPEE as the OPERATION. All other parameters, except RETURN-DATA and RETURN-CODE, are null.

Sample Calls

COBOL Call

```
.
COPY HRNHLWAC
COPY HRNHLLIST
.
.
CALL 'HRNHLLTM' USING HRNHLLWA 'STOPEE'
      HRN-NULL-LIST HRN-NULL-LIST HRN-NULL-LIST HRN-NULL-LIST
HRN-RETURN-DATA HRN-RETURN-CODE.
.
```

Non Re-entrant Assembler Call

```
.
CALL HRNHLLTM, (HRNHLLWA, C 'STOPEE',                                X
               0, 0, 0, 0, DATA, CODE), VL
.
.
DATA DS CL159      REASON CODE FROM HRNHLLTM.
CODE DS F         REASON CODE FROM HRNHLLTM
.
```

Returned Values

Return Codes	Description
0	STOPEE request succeeded.
7	<p>STOPEE request failed. The first four bytes of the RETURN-DATA parameter are set to the reason code. Refer to Call Level Interface Reason Codes on page 226.</p> <p>The special reason code of 144 indicates that the Execution Environment is still active with other users. Refer to Chapter 17, Multiple-Session Execution Environments in Batch, on page 255 for an explanation of why this would be an acceptable reason code.</p>

Starting the Session – STARTSS

Syntax

For the syntax of all Call Level Interface functions, see [Call Level Interface Specification on page 207](#) and [HRNHLLTM Module Parameters on page 209](#).

Calling Parameters

After an Execution Environment is started or located, your client program starts the session by specifying STARTSS as the OPERATION parameter. The OPERAND parameter is ignored. Overrides to the session parameters are supplied through the variable length PARM parameter.

Sample Calls

COBOL Call

```
.
COPY HRNHLWAC
COPY HRNHLLIST
.
.
CALL 'HRNHLLTM' USING HRNHLWAC 'STARTSS'
      HRN-NULL-LIST HRN-SESSPARM HRN-NULL-LIST HRN-NULL-LIST
      HRN-RETURN-DATA HRN-RETURN-CODE.
.
```

Non Re-entrant Assembler Call

```
      CALL      HRNHLLTM, (HRNHLWAC, C' STARTSS ',          X
      0, PARM, 0, 0, DATA, CODE), VL
.
.
PARM      DC  AL2(15), C' U=UUUUU, P=PPPPP '
DATA      DS  L159              REASON CODE FROM HRNHLLTM
CODE      DS  F                  RETURN CODE FROM HRNHLLTM
.
```

Returned Values

Return Codes	Description
0	STARTSS request succeeded. The first eight bytes of the RETURN-DATA parameter are set to the name of the TIBCO Object Service Broker user ID (in the assembler example, it is set to 'UUUUU').
4	STARTSS request failed. The first four bytes of the RETURN-DATA parameter are set to the reason code. Refer to Call Level Interface Reason Codes on page 226 .

Advanced STARTSS BATCH Usage

To interface with certain kinds of third-party environments, you can create a multi-session non-blocking batch Execution Environment by specifying an environmental wait routine specific to the third-party environment. Using the assembler interface, you can pass the address of a session context work area to the environmental wait routine by specifying an exit descriptor as the ninth parameter to HRNHLLTM. The exit descriptor is mapped by the assembler DSECT called HRNXD and distributed in the MACRO data set. It is of type SESHANDL.

Additional Information

For an example of setting up an environmental wait routine, refer to “Driver for testing Call Level Interface in Batch or TSO (Assembler)” in the member S6BCAPID in the ASM data set distributed with TIBCO Object Service Broker. For a complete description of this facility, refer to [Chapter 17, Multiple-Session Execution Environments in Batch, on page 255](#).

Stopping the Session – STOPSS

Syntax

For the syntax of all Call Level Interface functions, see [Call Level Interface Specification on page 207](#) and [HRNHLLTM Module Parameters on page 209](#).

Calling Parameters

When your client program has no active streams, it stops the session by specifying STOPSS as the OPERATION parameter. All other parameters, except RETURN-DATA and RETURN-CODE, are null.



- After stopping the session, the client program can subsequently start another session, possibly with different session parameters.
- If no sessions exist, the client program can stop the Execution Environment.

Sample Code

COBOL Call

```
COPY HRNHLWAC
COPY HRNHLLTM
.
CALL 'HRNHLLTM' USING HRNHLWAC 'STOPSS'
HRN-NULL-LIST HRN-NULL-LIST HRN-NULL-LIST HRN-NULL-LIST
HRN-RETURN-DATA HRN-RETURN-CODE.
.
```

Non Re-entrant Assembler Call

```
.
CALL HRNHLLTM, (HRNHLWAC, C' STOPSS', 0,
0, 0, 0, DATA, CODE), VL
.
DATA DS CL159 REASON CODE FROM HRNHLLTM
CODE DS F RETURN CODE FROM HRNHLLTM
```

Returned Values

Return Code	Description
0	STOPSS request succeeded.
9	STOPSS request failed. The first four bytes of the RETURN-DATA parameter are set to the reason code. Refer to Call Level Interface Reason Codes on page 226 .

Starting a Transaction – STARTTR

Syntax

For the syntax of all Call Level Interface functions, see [Call Level Interface Specification on page 207](#) and [HRNHLLTM Module Parameters on page 209](#).

Calling Parameters

Your client program starts a transaction by specifying STARTTR as the OPERATION parameter. All other parameters, except the RETURN-DATA and RETURN-CODE parameters, are null.



- Only one transaction can be active at a time.
- The transaction uses the transactional characteristics to determine the mode of table access and which rules libraries to use when calling a rule.
- After starting a transaction, you can call a rule, commit, roll back, or end the transaction. In addition, you can start another transaction (which starts at a new stream level) and modify its characteristics if required, temporarily suspending the current transaction.

What Limits the Number of Transactions

The maximum nesting level of active streams is limited by the session parameter TRANMAXNUM. If the Call Level Interface is used to call a rule that EXECUTES another rule, the number of active streams plus the number of nested EXECUTES is limited by TRANMAXNUM.

Sample Calls

COBOL Call

```
COPY HRNHLWAC
COPY HRNHLIST
.
CALL 'HRNHLLTM' USING HRNHLLWA 'STARTTR',
    HRN-NULL-LIST HRN-TRANPARAM HRN-NULL-LIST HRN-NULL-LIST
    HRN-RETURN-DATA HRN-RETURN-CODE.
.
```

Non Re-entrant Assembler Call

```
.
CALL HRNHLLTM,(HRNHLLWA,C 'STARTTR',
    0,PARAM,0,0,DATA,CODE),VL
.
.
DATA DS C159      REASON CODE FROM HRNHLLTM
CODE DS F         RETURN CODE FROM HRNHLLTM
.
```

Returned Values

Return Codes	Description
0	STARTTR request succeeded.
10	STARTTR request failed. The first four bytes of the RETURN-DATA parameter are set to the reason code. Refer to Call Level Interface Reason Codes on page 226 .

See Also *TIBCO Object Service Broker Parameters* about Execution Environment parameters.

Modifying Transactional Characteristics

Calling Parameters

Your client program can modify transactional characteristics by specifying the characteristics, for example UPDATE, as transaction parameters to the STARTTR operation. All other parameters, except RETURN-DATA and RETURN-CODE, are null.

What are the Transactional Characteristics?

The transactional characteristics are:

- BROWSE and UPDATE
- TEST/NOTEST
- SEARCH=*path*

path can be one of S, I, or L, using the variable string notation (the length is one).

- LIBRARY=*libraryname*

libraryname can be the name of a local library accessible to the TIBCO Object Service Broker user ID of the session, using the variable string notation (the maximum length is eight).

These are also session parameters. Refer to [Parameters for Your Session on page 26](#) for more information.

When Can the Modifications Be Made?

The transactional characteristics of a stream can be specified only when a transaction is started.

What is the Inheritance of Transactional Characteristics?

A stream inherits the values of the transactional characteristics from its parent. When the stream is the first stream in the session, the transactional characteristics are inherited from the session. In all other cases, the parent is the transaction at the lower transaction nesting level.

Sample Calls

COBOL Call

The following COBOL call sets the transactional characteristic defined in HRN-TRANPARAM:

```
COPY HRNHLWAC
COPY HRNHLIST
.
01  HRN-TRANPARAM.
    05  TRANPARMLEN          PIC9(4) USAGE COMP-4 VALUE 36
    05  TRANPARMDAT          PIC x(36)
        VALUE 'LIBRARY=TEST1,UPDATE,SEARCH=L,NOTEST'
.
CALL 'HRNHLLTM' USING HRNHLLWA 'STARTTR'
    HRN-NULL-LIST HRN-TRANPARAM HRN-NULL-LIST HRN-NULL-LIST
    HRN-RETURN-DATA HRN-RETURN-CODE.
.
```

Non Re-entrant Assembler Call

```
.
CALL HRNHLLTM,(HRNHLLWA,C'STARTTR',                                X
    0,TRANPARAM,0,0,DATA,CODE),VL
.
PARM  DC AL2(36),C'LIBRARY=TEST1,UPDATE,SEARCH=L,NOTEST'
*      TRANSACTION START LIST
DATA  DS      REASON CODE FROM HRNHLLTM
CODE  DS F     RETURN CODE FROM HRNHLLTM
.
```

Returned Values

Return Codes	Description
0	STARTTR request succeeded.
10	STARTTR request failed. The first four bytes of the RETURN-DATA parameter are set to the reason code. Refer to Call Level Interface Reason Codes on page 226 .

Ending a Transaction – STOPTR

Syntax

For the syntax of all Call Level Interface functions, see [Call Level Interface Specification on page 207](#) and [HRNHLLTM Module Parameters on page 209](#).

Calling Parameters

Your client program ends a transaction by specifying STOPTR as the OPERATION parameter. You can explicitly specify the COMMIT (the default if neither is specified) or ROLLBACK parameters to STOPTR. All other parameters, except RETURN-DATA and RETURN-CODE, are null.



- A transaction can be ended only if it was started and is currently active.
- Ending a transaction implies committing uncommitted data and dropping locks acquired during the course of the transaction.

Sample Calls

COBOL Call

```
COPY HRNHLWAC
COPY HRNHLLTM
.
CALL 'HRNHLLTM' USING HRNHLWAC 'STOPTR'
HRN-NULL-LIST HRN-NULL-LIST HRN-NULL-LIST HRN-NULL-LIST
HRN-RETURN-DATA HRN-RETURN-CODE.
```

Non Re-entrant Assembler Call

```
.
CALL HRNHLLTM, (HRNHLWAC, C'STOPTR',                                X
0,0,0,0,DATA,CODE),VL
.
DATA DS CL159                REASON CODE FROM HRNHLLTM
CODE DS F                    RETURN CODE FROM HRNHLLTM
.
```

Returned Values

Return Codes	Description
0	STOPTR request succeeded.
12	STOPTR request failed. The first four bytes of the RETURN-DATA parameter are set to the reason code. Refer to Call Level Interface Reason Codes on page 226 .

Calling a Rule – CALLRULE

Syntax

For the syntax of all Call Level Interface functions, see [Call Level Interface Specification on page 207](#) and [HRNHLLTM Module Parameters on page 209](#).

Calling Parameters

Your client program calls a rule by specifying CALLRULE as the OPERATION parameter and a string consisting of the rule name as the PARM parameter. If the rule takes arguments, the rule name is followed by the values for the arguments in parentheses. The DATA-IN (the fifth) and the DATA-OUT (the sixth) parameter are specified as described below. The RETURN-DATA (the seventh) parameter is set to any message that is returned by ENDMSG when the RETURN-CODE parameter is 0.



- When your client program has started a transaction, it can call an entry rule and pass values for its arguments. The rule has no real restrictions and can use all TIBCO Object Service Broker facilities, that is, it can call other rules and use the EXECUTE, DISPLAY, TRANSFERCALL, COMMIT, and ROLLBACK statements.
- When the called rule completes, control is returned to the client program. The transaction, however, ends only with a STOPTR call.

DATA-IN and DATA-OUT Areas

The client program can pass DATA-IN and DATA-OUT areas to the session:

The rule can read the DATA-IN areas using MAP tables or tools such as [\\$GETENVCOMMAREA](#).

The rule can write to the DATA-OUT areas using MAP tables or the [\\$SETENVCOMMAREA](#) tool.

The DATA-IN and the DATA-OUT areas can have one of two formats and describe one or more blocks of storage. In all cases, the client program is responsible for allocating storage, and for storage blocks pointed to by their structure.

The DATA-IN and the DATA-OUT areas are mapped by the same structure. The structure is used to represent a list of blocks (or COMMAREAs). The structure contains a count of the number of blocks, location and size.

Format Types

Each block is in one of the following formats:

- Format 1: Represented by a pointer to the storage block
Format 1 works well in an assembler environment and permits segmentation of the storage for each separate block. It requires pointer manipulation.
- Format 2: Implicitly follows the structure
Format 2 is suited to COBOL applications, as the structure describing the blocks and the blocks themselves are passed as one large contiguous storage area. It does not require pointer manipulation.

Format 1

The storage blocks are in separate locations, and are pointed to by addresses. This technique is not well-supported by the COBOL language, but works well in assembler.

Element	Length (bytes)	Description
Count	4	Number of blocks of storage in this group.
Address1	4	Pointer to block #1.
Length1	4	Length of block #1.
...		
AddressN	4	Pointer to block #N.
LengthN	4	Length of block #N.

Format 2

The storage blocks are part of the area being passed. Use this implementation for COBOL.

Element	Length (bytes)	Description
Count	4	Number of blocks in this group.
0	4	Storage blocks are always defined in pairs, consisting of a full word of binary zeroes followed by a full word containing the length of the block. This is the first field of binary zero for the first storage block.
Length1	4	This is the second field containing the length of the first storage block.
...		Repeat the pair of binary zero and length for the 2nd through N-1th storage block.
0	4	This is the first field of binary zero for the N-th storage block.
LengthN	4	This is the second field containing the length of the N-th storage block.
Commarea1	length 1	First block
...		2nd through N-1th storage blocks
CommareaN	length N	Nth storage block

Accessing the Storage Areas

From your rules code, you can obtain the address of the DATA-IN block list by using the value of the field APIINHANDLE of the System Interpreted Table @SESSION(0). Similarly, you can obtain the address of the DATA-OUT block list using the value of the field APIOUTHANDLE of @SESSION(0). Access to the storage referenced by MAP tables is always granted and there is no need to register these addresses in the @MAP table. APIINHANDLE memory is accessible for reading and APIOUTHANDLE for reading and writing.

You can also use the \$GETENVCOMMAREA and \$SETENVCOMMAREA tools to access the blocks of storage in the DATA-IN and DATA-OUT areas.

Since the user client is responsible for allocating storage for the DATA-IN and DATA-OUT block lists, changing the value for APIINHANDLE and APIOUTHANDLE is not supported. The contents of the block lists can, however, be modified by rules code using MAP tables.

Sample Rule Code

The following rule fragment assigns the address of the DATA-OUT block list to local variable OUT, and the address of the DATA-IN block list to local variable IN:

```
GET @SESSION(0);
IN=@SESSION.APIINHANDLE;
OUT=@SESSION.APIOUTHANDLE;
```

Sample Calls

These examples call the ABC rule with the two arguments: 1 and SCR. They pass in one commarea via DATA-IN and expect up to 80 bytes of data to be returned in DATA-OUT.

COBOL Call

```
01  HRN-CALLABC.
    05  RULEPARMLEN          PIC 9(4) USAGE COMP-4 VALUE 12.
    05  RULEPARMDAT          PIC X(12) VALUE 'ABC(1, 'SCR')'.
01  HRN-DATA-IN.
    05  IN-NO-COMMAREA       PIC 9(9) USAGE COMP-4 VALUE 1.
    05  IN-ZEROES            PIC 9(9) USAGE COMP-4 VALUE 0.
    05  DATAINPARMLEN       PIC 9(9) USAGE COMP-4 VALUE 27.
    05  DATAINPARMDAT       PIC X(49)
                                VALUE 'CALLING OSB FROM COBOL II'.

01  HRN-DATA-OUT.
    05  OUT-NO-COMMAREA      PIC 9(9) USAGE COMP-4 VALUE 1.
    05  OUT-ZEROES           PIC 9(9) USAGE COMP-4 VALUE 0.
    05  DATAOUTPARMLEN      PIC 9(9) USAGE COMP-4 VALUE 80.
    05  DATAOUTPARMDAT      PIC X(80) VALUE SPACES.

    :
    :
    CALL 'HRNHLLTM' USING HRNHLLWA 'CALLRULE'
    HRN-NULL_DATA HRN-CALLABC HRN-DATA-IN HRN-DATA-OUT
    HRN-RETURN-DATA HRN-RETURN-CODE.
```

Non Re-entrant Assembler Call

CALL	HRNHLLTM, (HRNHLLWA, C 'CALLRULE ' ,				X
	0 , CALLABC , DATAIN , DATAOUT , DATA , CODE) , VL				
.					
.					
CALLABC	DC	AL2(12) , C 'ABC(1, ' 'SCR' ') '			
DATAIN	DC	F(1) , A(DATAINC) , F(50)			
DATAINC	DC	CL28 'CALLING OSB FROM ASSEMBLER '			
DATAOUT	DC	F(1) , A(DATAOUTC) , F(80)			
DATAOUTC	DS	CL80			
DATA	DS	CL159	REASON CODE FROM HRNHLLTM		
CODE	DS	F	RETURN CODE FROM HRNHLLTM		

Return Values

Return Codes	Description
0	CALLRULE request succeeded.
11	CALLRULE request failed. The first four bytes of the RETURN-DATA parameter are set to the reason code. Refer to Call Level Interface Reason Codes on page 226 .

Non-Zero Value Returned

Whenever a non-zero value is returned in RETURN-CODE, RETURN-DATA contains either a reason code or an error message. The reason code is indicated by binary zeros in the first byte of RETURN-DATA; otherwise, RETURN-DATA contains a short error message similar to the ENDMSG that is returned to the workbench.

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

TIBCO Object Service Broker Programming in Rules about writing rules.

TIBCO Object Service Broker Shareable Tools about the tools and System Interpreted Tables.

Chapter 17 **Multiple-Session Execution Environments in Batch**

This chapter describes how to use multiple-session environments in batch.

Topics

- [Starting Multiple-Session Execution Environments in Batch, page 256](#)
- [Specifying an Environmental Wait Routine, page 257](#)
- [STARTEE Call, page 259](#)
- [STARTSS Call, page 261](#)
- [Sample Programs, page 262](#)

Starting Multiple-Session Execution Environments in Batch

You may want to allow multiple copies of a client program to use the Call Level Interface to run in the same address space as a batch Execution Environment. For example, if you are using a TP monitor other than CICS or IMS TM, you could find this beneficial.

What Facility Is Available?

With the Call Level Interface, you can specify an environmental wait routine specific to your platform. The environmental wait routine is specified as a user exit routine in an optional ninth parameter to the STARTEE or to the STARTSS interface function call.

Sample programs are provided to assist you in this specification of an environmental wait routine. These programs are described in [Sample Programs on page 262](#).

Implementation Guidelines

Here are some guidelines for implementing multiple-session Execution Environments:

- Write a startup transaction that is executed only once and is executed before any other client program begins execution.

Consider using the facilities of your TP monitor to schedule this transaction during startup. For example, in a CICS environment, you would do this through the use of the PLT table.

- Issue the STARTEE call normally from your client programs. They pass a ninth parameter to the STARTSS call (which is documented below).
- Invoke a shutdown transaction to stop the Execution Environment after all client application programs are terminated.

Consider using the facilities of your TP monitor to perform this on shutdown of the TP system. For example, in a CICS environment, you would do through the use of the PLTSD= parameter in the SIT table.

Specifying an Environmental Wait Routine

User exit routines are implemented by the use of an Exit Descriptor supplied as an optional ninth parameter to the STARTEE and STARTSS interface function calls. This optional ninth parameter is mapped by the assembler copybook HRNXD provided in the distribution MACRO data set.

Listing of the HRNXD Copybook

Field Name	Format	Explanation
HRNXDTYP	CL8	Indicate the type of user exit to be implemented. The required type is identified below for each user exit routine. The exit can be either SESENVWT or SESHANDL.
HRNXDNAM	CL8	Name of user exit routine.
HRNXDUSR	A	User field. Can be used to anchor common storage used by the exit routine.
HRN#INTT	H	Number of Interpreter Tasks—returned field.
HRN#SMFT	H	Number of SMF Writer Tasks—returned field.
HRNXDRS2	A	Reserved—must be 0.
HRNXDNXD	A	Reserved.
HRNXDRS3	CL8	Reserved—must be 0.
HRNXDRS4	XL2	Reserved—must be 0.
HRNXDXSZ	H	EXIT Block Extension size—maximum is 3K (3072 bytes).
HRNXDEP	A	Address of user exit routine (Entry Point Address).

User Exit Types Supported

Two user exit types are supported:

SESENVWT	Used to specify the entry point to the environmental wait routine.
SESHANDL	Used to pass the client's session environment to the environmental wait routine.

STARTEE Call

Behavior of STARTEE

The first time a STARTEE call is executed, the Execution Environment is started in the same address space as your client program. An internal counter, EECNT, is set to one. You can use the ninth parameter to supply the Execution Environment with the address of your user-written environmental wait routine.

Subsequent STARTEE calls cause the first parameter, HRNHLLWA, to be updated with internal information established by the first STARTEE call. The internal counter EECNT is incremented by one.

Storage of the User Routine Address

It is important to note that only on the first STARTEE call is the address of the environmental wait routine stored in the Execution Environment. Subsequent calls to STARTEE to locate the Execution Environment with this ninth parameter return only the contents of the HRNXDUSR field with the value it was set to on the initial STARTEE call. It is not necessary for your user application programs (client programs) to supply this optional ninth parameter.

Behavior in the Sample Programs

In the sample programs provided with TIBCO Object Service Broker, the initial STARTEE call is performed from a startup transaction program. Refer to [Sample Programs on page 262](#) for a description of these sample programs.

Assembler Example

The following excerpt of assembler code illustrates the STARTEE call, with some explanation following the example:

```
USEREXIT  CSECT ,
          ..
          ..
          XC      HRNXD(HRNXDSIZ),HRNXD      CLEAR THE HRNXD DSECT
          MVC      HRNXDTYP,=CL8'SESENVWT'    INFORM EE OF THE EXIT TYPE
          MVC      HRNXDNAM,=CL8'USEREXIT'    OUR PROGRAM NAME
          LA       R1,ENVWAIT                THE ADDRESS OF OUR ENTRY
          ST       R1,HRNXDEP                POINT IN OUR CODE
          OI       HRNXDEP,X'80'              SET HIGH ORDER BIT
          CALL     HRNHLLTM,(HRNHLLWA,STARTEE,BATCH,RGNPARM,0,0,RETDATA,      x
          RETCODE,HRNXD),VL,MF=(E,CALLIST)
          RETURN                                RETURN TO TP MONITOR
          ..
          ..
          ..
ENVWAIT   DS      0H                        CLIENT PROGRAM WAIT
*
          ROUTINE CODE HERE
```

Explanation of Values Provided for the Example

Field Name	Format	Value	Explanation
HRNXDTYP	CL8	SESENVWT	Indicates the exit routine is to be implemented.
HRNXDNAM	CL8	user program name	User-specified.
HRNXDUSR	A	A(eehandle)	This can be an address of some common storage for your environmental wait routine. It is returned to any caller of the STARTEE call that provides the optional ninth parameter.
HRNXDEP	A	A(EXIT RTN)	This must be the address of the Entry Point of your exit routine. The high order bit must be set on.

STARTSS Call

Purpose of STARTSS

You can use the Exit Descriptor in the STARTSS call to pass additional information to the environmental wait routine you specified in the STARTEE call. This additional information varies for each type of TP monitor or multiple-session environment you are implementing. As a guideline, it is likely a copy of your original save area upon entry to your client application, or some other control block that is required by the TP monitor to re-establish its environment prior to executing its own version of a wait macro.

Behavior in the Sample Programs

In the sample programs provided with TIBCO Object Service Broker, CICS is used as the TP monitor, even though a native interface exists for a CICS environment, that is, the OPERAND parameter is set to BATCH, not CICS on the STARTEE call.

In this particular environment CICS has a Control Block mapped in assembler by the DFHEISTG macro. This is the first-level save area for a user program. It also contains important pointers to other control blocks that CICS requires when you issue a CICS request (Command Level Call). In the sample, it is a copy of this Control Block that is to be passed to the environmental wait routine.

Explanation of the Possible Values

Field Name	Format	Value	Explanation
HRNXDTYP	CL8	SESHANDL	Indicates the environmental wait routine is to be implemented.
HRNXDNAM	CL8	user program name	User-specified.
HRNXDUSR	A	A(eehandle)	<p>The address stored in this field is passed to the environmental wait routine in Register 0 (zero), when it is called.</p> <p>It should be a Control Block that you use to re-establish the environment of the calling client program.</p>

Sample Programs

Programs Provided

Four programs are provided as a guideline for you to implement your own environmental wait routine. They can be found in the ASM and COBOL data sets distributed with TIBCO Object Service Broker. These programs are:

- S6BEWTIN
- S6BEWTSD
- S6BEWTSS
- COBCAPI3



These sample programs use CICS as the host TP monitor even though a standard CICS interface exists for this.

S6BEWTIN

This is an initialization transaction program to bring up the Execution Environment and supply the address of the environmental wait routine (also located in this program). A significant portion of this program involves setting up the ninth parameter to the STARTEE call.

Must be Resident to TP Monitor

As this program contains the environmental wait routine it is extremely important that it be marked resident to your TP monitor. This insures that the code pointed to by HRNXDEP (the Entry Point of your wait routine) does not relocate in memory. It is also *mandatory* that the environmental wait routine be written as *reentrant*.

You can ensure that the environmental wait routine is resident by having the initialization program use CSA storage to LOAD the exit routine there, or use other facilities possibly available in your TP environment, such as making the program resident.

S6BEWTSD

This transaction program is provided as the shutdown program. It should be executed as the last program to call TIBCO Object Service Broker. Failure to execute this program likely causes a System Abend Code A03.

Typically, this program would reside in your shutdown table. Its purpose is to tell the Execution Environment to shutdown. It first performs a STARTEE to initialize the HRNHLLWA work area, and performs the number of STOPEE calls required according to the program logic.

Program Logic

The Execution Environment maintains one internal counter, EECNT, for the number of STARTEE calls issued. When the shutdown program (S6BEWTSD) is executed, under normal circumstances there is one outstanding STARTEE call, from the initialization program (S6BEWTIN), plus one STARTEE call from this program. Therefore, the shutdown transaction must issue at least two STOPEE calls to quiesce the Execution Environment. The shutdown transaction contains logic to catch the situation where a user application program fails to issue a STOPEE call. In this case, extra STOPEE calls are issued and a warning message appears.

S6BEWTSS

This assembler program would assist in setting up the Exit Descriptor (ninth parameter) for the STARTSS call in your COBOL client programs. It takes two input parameters. As part of the setup, a copy of the COBOL's original first-level save area (the CICS DFHEISTG Control Block) is made, and its address is placed in the HRNXDUSR field of the data area named HRNXD further down in the code.

Calling Requirements

- You must call this routine from the mainline of your COBOL program, before the STARTSS call.
- You must not call it from within a subroutine as it is impossible to determine where the original save area is located.
- If you use an external routine that is link-edited with your main program to issue the STARTSS call, you must pass the HRNXD data field as a parameter to that external routine. You must first initialize the data field in your mainline code of the originating program with the call to S6BEWTSS.
- If the external routine is executed via the TP monitor's LINK facility (such as a CICS command-level **EXEC CICS LINK** call), it should be sufficient to place the S6BEWTSS call in the external routines mainline code.

Example

COBOL programmers do not need to be concerned with individual data elements within the two following data areas, but they must provide adequate storage as noted by the following example. Modify this example program according to the TP monitor conventions for your TP environment.

```

*
* The following is used as the ninth parameter on the STARTSS Call.
*
    01      HRNXD          PIC X(48).
      ..
      ..
* The following is used as a copy of the CICS DFHEISTG Control Block.
*
    01      HRNEISTG PIC X(248).
      ..
      ..
    CALL 'S6BEWTSS' USING HRNXD, HRNEISTG.
      ..

```

COBCAPI3

This is a COBOL program used to execute a TIBCO Object Service Broker rule.

Chapter 18 **TIBCO Object Service Broker SDK (C/C++) Server**

This chapter describes the TIBCO Object Service Broker SDK server for C and C++.

Topics

- [Introducing TIBCO Object Service Broker SDK \(C/C++\), page 266](#)
- [Execution Environment Considerations, page 267](#)
- [Additional Requirements for CICS Execution Environments, page 269](#)

Introducing TIBCO Object Service Broker SDK (C/C++)

The TIBCO Object Service Broker SDK (C/C++) client is an extension of the Call Level Interface. It extends the interface beyond the boundaries of the Execution Environment. This chapter explains how to make the SDK (C/C++) available to others to access your copy of TIBCO Object Service Broker.

Information on the Other Interfaces

Depending on your environment, you use one of the following:

External Environment	Interface	Refer to
Programs running in the same batch, TSO, or CICS environment as TIBCO Object Service Broker	Call Level Interface	Chapter 14, Introduction to the Call Level Interface, page 199. Chapter 15, Preparing the Environment, Analyzing Returned Values, and Modifying Changes, page 219. Chapter 16, Call Level Interface Functions, page 233. Chapter 17, Multiple-Session Execution Environments in Batch, page 255.
Java programs	SDK (Java)	Chapter 20, Using TIBCO Object Service Broker SDK (Java), page 303.

Required Parameters

The following Execution Environment parameters must be set when setting up SDK (C/C++). Refer to [Set the Required Execution Environment Parameters on page 267.](#)

- CLIMSGLENMAX – Maximum length of an SDK (C/C++) or SDK (Java) message that can be sent or received between clients and servers. The length includes the message and control information.
- EENAME – Name of the Execution Environment to be used to run the session.
- STANDBYNUM – The number of Execution Environment standby sessions.

See Also *TIBCO Object Service Broker for z/OS Installing and Operating* for information about installing the TIBCO Object Service Broker SDK (C/C++) server.

Execution Environment Considerations

Preparatory Steps

As well as the tasks listed in [Preparing to Start or Locate the Execution Environment on page 220](#), complete the following tasks before you start a Native Execution Environment or CICS Execution Environment.

Task A Enable and Initialize National Language Support (NLS)

The TIBCO Object Service Broker SDK (C/C++) for z/OS requires that the NLS feature of TIBCO Object Service Broker be enabled and initialized in the Execution Environment.

For detailed information on how to enable and configure NLS in the Execution Environment, refer to *TIBCO Object Service Broker National Language Support*.

Task B Set the Required Execution Environment Parameters

You must set certain parameters for your Execution Environment:

- CLIMSGLENMAX – Specify the maximum length of an SDK (C/C++) message

Set this parameter to limit the length of the messages between the client and the Execution Environment. This length includes the SDK (C/C++) message and the control information. The maximum you can specify is 32 MB.

- EENAME – Identify the TCP/IP socket

Set this parameter to the eight-byte ID of the IP address of the TCP/IP socket. If the same Native Execution Environment is to support VTAM terminals, the VTAM definitions should have the same APPLID.

- STANDBYNUM – Set the Number of Standby Sessions

Set this parameter to the number of standby sessions you want initialized in this Execution Environment region:

TSO and Batch – Set STANDBYNUM=1

Native and CICS – Set STANDBYNUM to the anticipated maximum number of concurrent SDK (C/C++) clients in the CICS region, to a maximum of 4096

A Standby Session is tied to the SDK (C/C++) client for the duration of a session. Therefore the number of standby sessions controls the number of concurrent SDK (C/C++) client sessions.

To improve the performance of the SDK (C/C++) client/server sessions, consider increasing the number of Session Initiator Tasks (using the TASKINITNUM Execution Environment Region parameter). The number of SIN tasks is independent of the number of concurrent SDK (C/C++) client sessions.



For the CLIMSGLENMAX parameter, you can specify 0M for a special meaning of “no limit”. Use it with caution. Specifying 0M can cause storage constraint in the system.

See Also *TIBCO Object Service Broker Parameters* about Execution Environment parameters.

Task C If necessary, configure the Execution Environment for listening on a TCP/IP port

To communicate with an SDK (C/C++) client from Windows or Solaris, the Execution Environment in z/OS must be configured to listen on a TCP/IP port.

For detailed information on how to configure the Execution Environment, refer to *TIBCO Object Service Broker for z/OS Installing and Operating*.

Additional Requirements for CICS Execution Environments

SIT Parameter Requirements

To run the SDK (C/C++) session CICS background task under the SDK (C/C++) client session's user ID (**USERID1**) and to enable CICS surrogate user checking, you must start CICS with the following SIT parameters:

`DFLTUSER=USERID2, SEC=YES, XUSER=YES`

The following parameters are optional, although you can specify these if you are not implementing security for them at this time:

`XCMD=NO
XDCT=NO
XFCT=NO
XJCT=NO
XPCT=NO
XPPT=NO
XPSB=NO
XTRAN=NO
XTST=NO`

Specifying the CICS Session Background Task Transaction

To set up the SDK (C/C++), use the following background task:

- Transaction name: HCLI
- Program name: S6BCSCLI

Specifying RACF Definitions

When the TIBCO Object Service Broker CICS SDK (C/C++) server receives a CONNECT request from the SDK (C/C++) client, it starts a special type of HURN transaction (HCLI), which runs under **USERID2**, the value specified by the DFLTUSER parameter, from the z/OS console. This HURN transaction in turn starts the SDK (C/C++) background task by issuing:

`EXEC CICS START TRANSID('HCLI') userid('USERID1')`

USERID2 must be defined to your security system, as a surrogate of **USERID1** (with READ authority) as illustrated in the following RACF definition commands:

`RDEFINE SURROGAT USERID1.DFHSTART UACC(NONE) OWNER(USERID1)
PERMIT USERID1.DFHSTART CLASS(SURROGAT) ID(USERID2) ACCESS(READ)`

See Also [Selecting a TIBCO Object Service Broker CICS Client Program on page 66](#)

TIBCO Object Service Broker for z/OS Installing and Operating about installing the CICS component of TIBCO Object Service Broker.

TIBCO Object Service Broker Parameters about parameters.

Chapter 19 **Using TIBCO Object Service Broker SDK (C/C++)**

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

Topics

- [Overview of the TIBCO Object Service Broker SDK \(C/C++\), page 272](#)
- [SDK \(C/C++\) Functions, page 275](#)
- [Sample Application Using the SDK \(C/C++\), page 301](#)

Overview of the TIBCO Object Service Broker SDK (C/C++)

What Is the TIBCO Object Service Broker SDK (C/C++)?

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.

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 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 z/OS, in another work space on the same computer.

How Can It Be Used?

With the SDK (C/C++), you can 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 the programming language of your choice. 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 301](#).

Compiling and Running

1. Start your Data Object Broker.

For our example, we are using the following parameters:

```
COMMID=D364046@
NODENAME=A
```

2. Start a Native Execution Environment.

For our example, so that this Execution Environment connects to the Data Object Broker referenced in step #1., we use the following parameters:

```
TD=D364046@
MD=N364046@ /* This value is supplied in the call to the sample
program */
STANDBYNUM=2
CLIMSGLENMAX=1M
```

3. Copy the source code for your program to hlq.SOURCE(RCLISAMP).
4. Compile, bind and run your program using the JCL in the RCLIJCL member of the JCL data set provided with the SDK (C/C++) at installation.

Make sure that the IBM cataloged procedures CBCC, CBCB, and CBCG are accessible by this job.

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 283 .
	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 287 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.

Rule Calls, Session and Transaction Management

Name	Brief description	On page
cliProc	Serves as the main SDK (C/C++) entry point for processing STARTSS, STARTTR, CALLRULE, STOPTR, STOPSS, RESETSS, GETENDMSG, and SESSACTIVE requests.	277
cliExecTran	Performs transaction start, rule call, and transaction end as a single SDK (C/C++) call.	289

Code Page Setting and Error Retrieval

Name	Brief description	On page
cliSetCodepage	Sets an SDK (C/C++)/SDK (Java) code page.	291
cliErrorReasonDescr	Retrieves the textual description of an error reason code.	293

Commarea Helper Functions

This group of functions facilitate dataIn and dataOut commarea processing. All the functions work with the format described in [Calling a Rule – CALLRULE on page 249](#). Generally, these functions do not validate memory pointers passed as parameters.

Name	Brief description	On page
cliCommCreate	Allocates memory and formats it according to the commarea format.	294
cliCommCreate1	Allocates and formats a single-segment commarea.	294
cliCommDelete	Deletes a commarea created by cliCommCreate or by cliCommCreate1.	295
cliCommFormat	Formats memory according to the commarea format.	295
cliCommFormat1	Formats a single-segment commarea.	296
cliCommSegment	Retrieves a pointer to a commarea segment.	297
cliCommSegments	Retrieves the number of segments in a commarea.	297
cliCommSegSize	Retrieves the commarea segment size.	298
cliCommSize	Calculates the total commarea size.	298
cliCommSizeCalc	Calculates the size of a commarea for a given structure.	299
cliCommSizeCalc1	Calculates the size of a single-segment commarea, given the size of the segment.	299

C Macros

Name	Brief description	On page
LLCOPY_CSTR	Copies a zero-terminated string to a string with a two-byte length prefix.	300
LLCOPY_MEM	Copies a string with an explicitly specified length to a string with a two-byte length prefix.	300
LLDECLARE	Declares a string with a two-byte length prefix.	300
LLSETLEN	Sets a two-byte length prefix.	300
LLSTR	Returns a pointer to the text part of a string that has a two-byte length prefix.	300
LLSTRLEN	Returns the string length from a string that has a two-byte length prefix.	300

See Also [Appendix A, SDK \(C/C++\) and SDK \(Java\) Error Reason Codes, on page 373](#) for a list of error reason codes issued in relation to 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	On page
STARTSS	STARTSS – Start a Session.	280
STARTTR	STARTTR – Start a Transaction.	282
CALLRULE	CALLRULE – Call a Rule.	283
STOPTR	STOPTR – Stop a Transaction.	286
STOPSS	STOPSS – Stop a Session.	286
RESETSS	RESETSS – Drop a Connection to a Session.	287
GETENDMSG	GETENDMSG – Retrieve a Rules End Message.	287
SESSACTIVE	SESSACTIVE – Inquire Whether Session Is Active.	288

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 the names until the match is found. If the request name is not one of the above, 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 <i>operation</i> 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 <i>operation</i> 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 <i>operation</i> 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 <i>operation</i> 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 below.



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

STARTSS starts a new TIBCO Object Service Broker 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 24-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 EBCDIC. For valid values, refer to cliSetCodepage on page 291 .
	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).

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 (case-insensitive) parameters are:

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.

You use CLINODE to identify the TIBCO Object Service Broker monitor process (Windows or Solaris) or the Execution Environment (z/OS) on the network. See your TIBCO Object Service Broker administrator for the node name of the Execution Environment, or osMon in Windows or Solaris, to which you want to connect. For more information on the CLINODE parameter, refer to *TIBCO Object Service Broker Parameters*.

Make sure that your *session* parameter string contains 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.



Open Systems programs using the SDK (C/C++) also have available the CLIHOST and CLIPORT parameters as an alternative to CLINODE.

Use of a CLIHOST/CLIPORT pair on z/OS is not supported. It fails with a COMMFAILURE (195) error reason code.

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 <i>params</i> 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 (all characters are 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 abends or is stopped, STARTTR fails with a 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	<p>Points to the maximum length for the rules return value that is placed in the <i>retData</i> buffer, including the two-byte length prefix and the terminating zero.</p> <p>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 <i>operand</i> is NULL, no return value is stored.</p> <p>This parameter is In/Out. Refer to <i>operand</i> under Out below.</p>
	params	<p>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_MAXRULEEXPRLN in oscli.h).</p>
	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 <i>operand</i> 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 <i>retData</i> is passed through the <i>operand</i> 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).

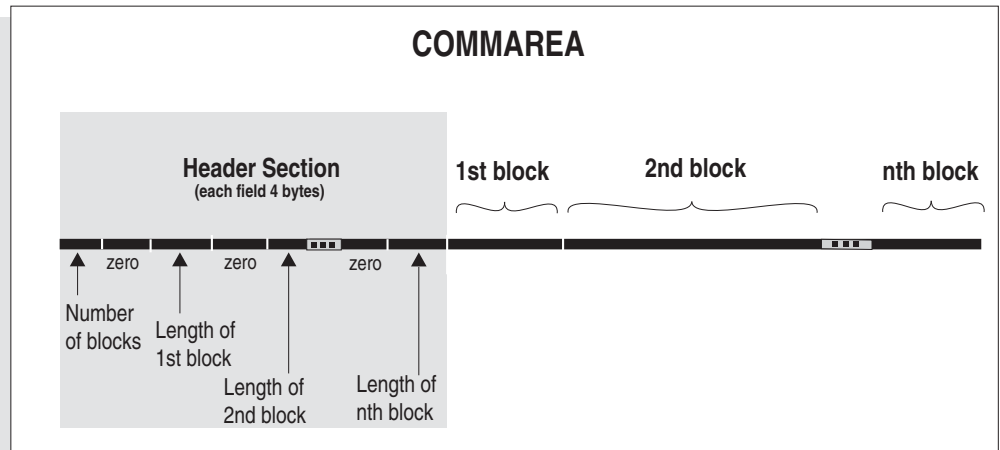
CALLRULE performs a rule call. Rule name and parameters are supplied in a textual form: RULENAME(PARAM1, PARAM2,...,PARAMn). The maximum length of this string is 514 (CLI_MAXRULEEXPRLen) excluding the length prefix. If a longer string is passed, CALLRULE fails with a CLI_RULEEXPRTOOLONG (3090) error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

If the session stops or abends, or no transaction was 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 transmitted to the application by your rule. You do this by properly reformatting the dataOut header, then, 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 stops, if it abends, or if no transaction was 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 abends or was stops, STOPSS fails with a CLI_CALLOUTOFSEQ (36) error reason code.

If there are transactions active within the session, STOPSS fails with a CLI_TRANSACTIONAL (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* is not previously passed to STARTSS or became corrupt, the operation fails with CLI_SESSINVALID(199). Refer to [cliProc on page 277](#) 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 TIBCO Object Service Broker rules end message is 148 (CLI_MAXENDMSGLEN in oscli.h), therefore, to accommodate a possible end message, 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 <i>retData</i> .
	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* is not previously passed to STARTSS or became corrupt, the operation fails with CLI_SESSINVALID(199). Refer to [cliProc on page 277](#) 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 it starts (due to network problems, Execution Environment abnormal terminations, and others).

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 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* is not previously passed to STARTSS or became corrupt, the operation fails with CLI_SESSINVALID(199). Refer to [cliProc on page 277](#) 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 282 .
ruleName	Rule call in the form of the string <code>RULE(PARAM1 , PARAM2 ,...PARAMn)</code> . For more detail, refer to CALLRULE – Call a Rule on page 283 .
retBufLen	Maximum length of the <i>retData</i> buffer. For more detail, refer to CALLRULE – Call a Rule on page 283 .
dataIn	<i>dataIn</i> commarea. For more detail, refer to CALLRULE – Call a Rule on page 283 .
dataOut	<i>dataOut</i> commarea. For more detail, refer to CALLRULE – Call a Rule on page 283 .

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 283 .
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 283 . If the call fails, the error reason code (four bytes) is stored in the <i>retData</i> 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 EBCDIC. The valid values for this field are the code page names shown in <i>TIBCO Object Service Broker National Language Support</i> .
----------	--



- 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.

Also, 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”.

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 280](#).

Initial value of the code page name is ISO8859-1 for Open Systems, and IBM-037 for z/OS.

cliSetCodepage stores, without validation, the code page name for future STARTSS cliProc requests (with 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 (with 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:

<code>reasonCode</code>	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 283](#)), 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.

<code>segmentSizes</code>	Array of block sizes.
---------------------------	-----------------------

Return Value:

None.

Comments

For more about the commarea format, refer to [CALLRULE – Call a Rule on page 283](#).

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:

<code>area</code>	Pointer to commarea memory.
<code>segmentSize</code>	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 283](#).

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 283](#)), 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 283](#)), the behavior is undefined.

cliCommSize

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

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

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 283](#)), 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++)



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

C Program

The sample program is available in the RCLISAMP member of the C data set distributed with TIBCO Object Service Broker SDK (C/C++).

The RCLIJCL member of the JCL data set distributed with TIBCO Object Service Broker SDK (C/C++) contains JCL to compile and run the sample program.

Rule Called by Program

The TC007113RU02 rule 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 the Program

The out from the 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 20 **Using TIBCO Object Service Broker SDK (Java)**

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

Topics

- [Overview of TIBCO Object Service Broker SDK \(Java\), page 304](#)
- [SDK \(Java\) Methods, page 308](#)
- [Session Object Methods, page 312](#)
- [SessionException Object Methods, page 327](#)
- [Misc Object Methods, page 330](#)
- [Sample Application Using the SDK \(Java\), page 336](#)

Overview of TIBCO Object Service Broker SDK (Java)

What Is the TIBCO Object Service Broker SDK (Java)?

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 TIBCO Object Service Broker SDK (C/C++), which is described in [Chapter 19, Using TIBCO Object Service Broker SDK \(C/C++\)](#), on page 271.

Information on the Other Interfaces

Depending on your environment, you use one of the following:

External Environment	Interface	Refer to
Programs running in the same batch, TSO, or CICS environment as TIBCO Object Service Broker	Call Level Interface	Chapter 14, Introduction to the Call Level Interface , page 199.
		Chapter 15, Preparing the Environment, Analyzing Returned Values, and Modifying Changes , page 219.
		Chapter 16, Call Level Interface Functions , page 233.
		Chapter 17, Multiple-Session Execution Environments in Batch , page 255.
C programs, C++ programs	SDK (C/C++)	Chapter 18, TIBCO Object Service Broker SDK (C/C++) Server , page 265.
		Chapter 19, Using TIBCO Object Service Broker SDK (C/C++) , page 271.

Requirements

Java Runtime

The SDK (Java) classes were built using the Java 1.4.1 compiler relying on APIs defined in version 1.1 of the Java platform. Therefore, according to the Java specification, the SDK (Java) methods can run on Version 2.0 and should “generally run on [the] 1.1 version of the Java virtual machine”.

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				SCROLL: P
COMMAND ==>				
KEY	COMPTYPE	COMPNAME	LOCALE_CP	
-----				-----
—	1 SELF		ENGL . IBM-037	
—	2 REMOTE		ENGL . IBM-037	

PFKEYS: 1=HELP 5=FOUND 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 336](#).

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 user ID 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 314 . The value is 514.
MAXENDMSGLEN	The maximum length of a rules end message. For more information, refer to endMessage on page 317 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:

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

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

Session Class

Method	Brief description	On page
Session	The class constructor.	312
start	Starts an SDK (Java) session.	321
stop	Stops an SDK (Java) session.	324
reset	Drops a connection to a session.	320
startTrans	Starts a transaction.	323
stopTrans	Stops the currently active transaction, committing or rolling back the changes.	325
call	Calls a rule.	314
transNestLevel	Returns the transaction nesting level of the session.	325
isActive	Returns the activity status of the session.	320

Method	Brief description	On page
endMessage	Returns the end message from the last rule called within the session.	317
userId	Returns the user ID of the session.	326
shutdown	Stops all transactions (committing or rolling back the changes), and stops the session regardless of the errors encountered.	321
execTran	Equivalent to executing a sequence of startTrans, call, and stopTrans methods. The changes are committed if the rule call succeeds and rolled back if it throws an exception.	318

SessionException Class

Name	Brief description	On page
SessionException	The SessionException class constructor.	327
rc	Returns an SDK (Java) operation error code.	329
reasonCode	Returns an error reason code for an SDK (Java) error.	328
errorReasonDescr	Returns a textual description of a particular SDK (Java) error reason code.	328

Refer to [Appendix A, SDK \(C/C++\) and SDK \(Java\) Error Reason Codes](#), on [page 373](#) for a list of error reason codes issued by the SDK (Java).

Misc Class

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

Name	Brief description	On page
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.	334
readShort	Writes, into a byte array, 2 bytes of a big-endian byte representation of a value of type short.	334
readShort	Reads 4 bytes from a byte array and returns a value of type int according to what these bytes represent in big-endian format.	334
readShort	Writes, into a byte array, 4 bytes of a big-endian byte representation of a value of type int.	334

This group of functions facilitate dataIn and dataOut commarea processing. These functions work with commareas of the format described in [call on page 314](#). Generally, these functions do not validate input parameters.

Name	Brief description	On page
commSizeCalc	Calculates the number of bytes needed for a commarea with the specified structure.	333
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	330
commFormat	Formats a byte array according to the commarea specification.	330
commSegmentInd	Returns the offset of a specified segment in a commarea.	331
commSegSize	Returns the size of a given commarea segment.	332

Name	Brief description	On page
commSize	Calculates the number of bytes in a commarea according to its header.	332
commSegments	Returns the number of segments in a commarea according to its header.	331

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 <i>TIBCO Object Service Broker National Language Support</i> .



- 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.

Also, 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 315).
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 <i>ruleRetValue</i> is null, no return value is stored.
ruleRetValueStart	The index where the SDK (Java) should start storing the return value in the <i>ruleRetValue</i> array.
ruleRetValueMaxLen	The maximum number of bytes available for the rules return value in the <i>ruleRetValue</i> array. If <i>ruleRetValueMaxLen</i> 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.MAXRULEEXPRLLEN), both forms throw an exception with an error reason code of Session.RULEEXPRTOOLONG (3090).

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

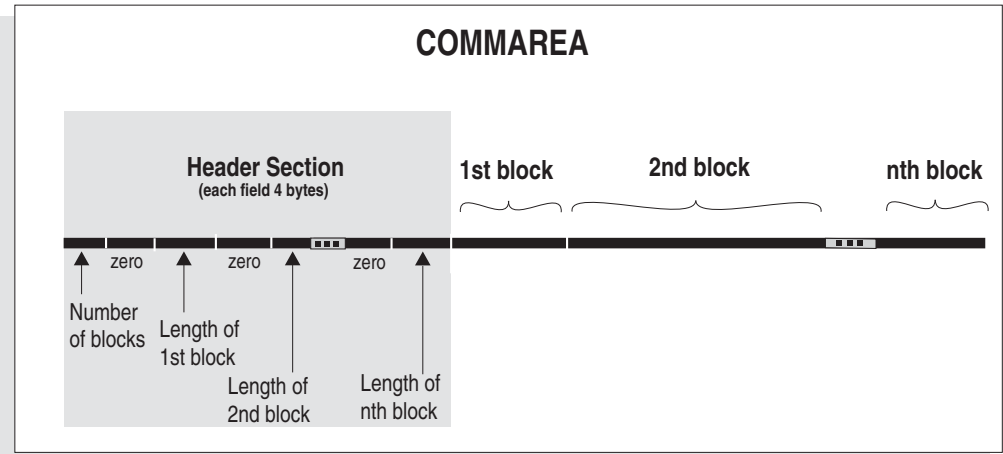
The two forms of the call method differ in how they handle the rules return value:

- 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 321](#)), 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 *ruleRetValueStart* 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 Object Service Broker Managing Data*.

When a rule completes successfully, 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. Then, before transmitting data back, the Execution Environment reevaluates the dataOut header to determine the correct number of bytes to send back to the application. You cannot 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 rule 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 tables.

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 rule 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 rule 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 rule 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 rule 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.
- When used 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
```

Parameters:

transParams	The transaction parameter string. Refer to startTrans on page 323 .
--------------------	---

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 314).
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 <i>ruleRetValue</i> 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 either 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 rule call

The only phase able to interrupt the sequence is the [startTrans](#) phase. After that phase has succeeded, the failure of the rule 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 abended, `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 312 .

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).) If not, 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 = <code>SessionException.STARTSS_FAILED (4)</code> 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 <code>SessionException.CALLOUTOFSEQ (36)</code> error reason code. To bring a session back to the inactive state, issue a stop , a reset , or a shutdown method call.
See Also	<i>TIBCO Object Service Broker Parameters</i> 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 = <code>SessionException.STARTTTR_FAILED (10)</code> 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 abends or is stopped, 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 was 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 was 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 rule 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 373](#).

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 373](#).

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 are 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 are 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 are 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 are 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 are 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 are 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 are 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 are 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 are 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 are thrown.

Sample Application Using the SDK (Java)

Compiling and Running the Sample Program

The sample program was compiled and executed with the following assumptions:

- The SDK (Java) .jar file and the source file JCLISAMP.java exist in the current directory.

You can find JCLISAMP.java in the cli.jar file. To copy JCLISAMP.java to the current directory, use the following command:

```
jar xf cli.jar JCLISAMP.java
```

- The Data Object Broker is running.
- A native Execution Environment, configured for TCP/IP, is connected to the Data Object Broker and listening on port 9068.
- NLS is set up.

Use the following commands to compile and run the sample program:

1. TSO OMVS

This starts the z/OS shell.

2. export CLASSPATH=".:./cli.jar:\$CLASSPATH"
3. javac JCLISAMP.java
4. java JCLISAMP clihost=os390a,cliport=9068 IBM-037

IBM-037 is the code page to be used.



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

See Also *TIBCO Object Service Broker National Language Support* about the @NLS1 table.

Sample Rule Called by a Program

This rule 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 ==>
TC007124RU02;

-----
TC007124TA01.TEXT = 'RULE "TC007124RU02" IS CALLED';      | 1
INSERT TC007124TA01;                                         | 2
CALL ENDMMSG('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
```

After the rule executes, 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

Chapter 21

Coding TIBCO Object Service Broker Access Statements

This chapter describes how to code TIBCO Object Service Broker access statements.

Topics

- [Overview, page 340](#)
- [Writing COBOL with TIBCO Object Service Broker Access Statements, page 342](#)
- [Coding the Access Statements, page 345](#)
- [Coding Considerations, page 346](#)

Overview

How to Access TIBCO Object Service Broker Data

You can access a TIBCO Object Service Broker table from a COBOL source program that contains embedded TIBCO Object Service Broker access statements or SQL statements. The COBOL program participates with TIBCO Object Service Broker as a client as opposed to a server as outlined in this manual.

If you have an existing COBOL program that accesses databases such as DB2, you can modify the program to access the TIBCO Object Service Broker database. If the COBOL program came as a package that normally accesses DB2 with SQL statements, you can use the program with a few modifications to access the TIBCO Object Service Broker database instead.

What if TIBCO Object Service Broker Table Access is Not Required?

If you want to use a COBOL program in TIBCO Object Service Broker that does not have to access TIBCO Object Service Broker tables, refer to [Chapter 14, Introduction to the Call Level Interface, page 199](#) to [Chapter 17, Multiple-Session Execution Environments in Batch, page 255](#).

Steps Required

Accessing TIBCO Object Service Broker from a COBOL program requires the following tasks:

- One of the following:
 - [Writing COBOL with TIBCO Object Service Broker Access Statements, page 342](#)
 - [Writing a COBOL Program with Embedded SQL Statements, page 352](#)
- [Preprocessing the Access Statements, page 368](#)
- [Preparing the Program, page 370](#)
- [Running the Program, page 372](#)

Samples Provided

Samples are provided showing COBOL programs with embedded TIBCO Object Service Broker and SQL access statements. The following table lists the samples that are shipped with TIBCO Object Service Broker in the COBOL and JCL data sets:

Member Name	Description
COBOSTMT	A COBOL program with TIBCO Object Service Broker access statements.
JCLOSTMT	JCL to: Preprocess the COBOL program that contains TIBCO Object Service Broker access statements, using the HLIPREPROCESSOR. Compile the processed program. Link-edit the object modules.
COBSQL	A COBOL program with SQL access statements.
JCLSQL	JCL to: Preprocess the COBOL program that contains SQL access statements, using the HLIPREPROCESSOR. Compile the processed program. Link-edit the object modules.

Writing COBOL with TIBCO Object Service Broker Access Statements

To access a TIBCO Object Service Broker table from your COBOL program you must add TIBCO Object Service Broker access statements. These statements are converted into valid COBOL statements by the TIBCO Object Service Broker preprocessor. This section provides an example of a COBOL program coded with TIBCO Object Service Broker access statements and the associated TIBCO Object Service Broker table definition. It also explains the statements you require for your program. For information on coding SQL statements in a COBOL program, refer to [Chapter 22, Coding SQL Access Statements, on page 351](#).

Sample COBOL Program

The following sample shows a COBOL program containing TIBCO Object Service Broker access statements. The following conventions are used in the example:

- Statements in mixed case letters starting with a dollar sign (\$) are TIBCO Object Service Broker access statements.
- Statements in uppercase letters are COBOL statements.



Parameter values must be typed in UPPERCASE.

Example

```
CBL MAP,RENT,NOSEQUENCE,TEST(SYM),THREAD,VBREF,OFFSET
IDENTIFICATION DIVISION.
PROGRAM-ID. COBOSTMT RECURSIVE.
AUTHOR. JOHN SMITH.
INSTALLATION. TIBCO Software Inc.
DATE-COMPILED.
*****
* Demonstration of HLI Preprocessor *
* A COBOL program with TIBCO Object Service Broker access statements *
* for TSO or batch TIBCO Object Service Broker sessions. *
*
* To execute this Cobol program, you must make an entry in the *
* OSB ROUTINES table such as: *
*      NAME           :COBOSTMT *
*      LANGUAGE       :LEPERSIST *
*      FUNCTION       :N *
*      LOADNAME       :COBOSTMT *
*
```



```

* Invoke the program via a RULE call:
*   CALL 'COBOSTMT';
*
* The program uses the table #ED_EMPLOYEES as its data source.
*
*****
ENVIRONMENT DIVISION.
*
DATA DIVISION.
*
WORKING-STORAGE SECTION.
*
* Variables in this section remain in their last state for
* every invocation of this subroutine within a run-unit /
* enclave.
*
77 USER-ID-77          PIC X(16) VALUE 'EDUC'.
77 LOCATION-77         PIC X(16) VALUE 'A'.
* Include Statement.
$ Include Huron;
* ***
* Define table Employee and translate non-Cobol names
* to acceptable Cobol names.
* ***
$ Define Table Employee = #ED_EMPLOYEES
$   Parameter USER-ID = USERID,
$   LOCATION-PARM = LOCATION
$   Field mgrno = MGR#,
$   zipcode = P_CODE;
*
LOCAL-STORAGE SECTION.
*
* Variables in this section are initialized at each invocation.
*
LINKAGE SECTION.
*
* Variables in this section are passed from/to caller on each
* invocation. Use ARGUMENTS table to define parameters to pass
* and the "PROCEDURE DIVISION USING parm1, parm2" etc to pass
* entries to the external routine as necessary. Each parameter
* should be a LINKAGE SECTION 01 entry.
*
PROCEDURE DIVISION.
MAINLINE-CODE SECTION.
$ Entry;
    DISPLAY 'ENTERED COBOSTMT COBOL PROGRAM'.
* Get and display names of selected employees.
* Set USER-ID-77 and LOCATION-77 to appropriate values
*   for your system
$ Forall Employee(USER-ID-77, LOCATION-77)
$   where mgrno = 84021 :
    DISPLAY EMPLOYEE-LNAME.
$ end;
EXIT-HERE.
    DISPLAY 'ABOUT TO EXIT COBOSTMT COBOL PROGRAM'.
    GOBACK.

```

Sample TIBCO Object Service Broker Table Definition

The following definition is of the table #ED_EMPLOYEES. This is the TIBCO Object Service Broker table used by the COBOL program with embedded TIBCO Object Service Broker access statements. (Only the first 80 columns are shown.)

Partial Definition of the #ED_EMPLOYEES Table

COMMAND==>						TABLE DEFINITION							
Table: #ED_EMPLOYEES						Type: TDS		Unit: EDUC		IDgen: N			
Parameter Name		Typ	Syn	Len	Dec	Class				Event Rule		Typ	Acc
-----		-	-	----	--	-				-----		-	-
- USERID		I	C	16	0	D							
- LOCATION		I	C	16	0	L							
Field Name		Typ	Syn	Len	Dec	Key	Ord	Rqd	Default		Reference		
-----		-	-	----	--	-	-	-	-----		-----		
- EMPNO		I	P	3	0	P							
- LNAME		S	C	22	0								
- POSITION		S	C	14	0								
- MGR#		I	P	3	0					MANAGER			
- DEPTNO		I	B	2	0								
- SALARY		Q	P	3	2								
- HIREDATE		D	B	4	0								
- ADDRESS		S	V	38	0								
- CITY		S	C	20	0								
- PROV		S	C	3	0								
- P_CODE		S	C	7	0								
-													
-													
PFKEYS: 3=END 12=CANCEL 22=DELETE 13=PRINT 14=FIELDS 21=DATA 2=DOC													

Coding the Access Statements

Where Do You Code the Access Statements?

Code the access statements in the Working Storage Section and Procedure Division of your COBOL program. Each line of an access statement must start with the dollar sign symbol (\$) in column 7, followed by the statement itself starting in column 12.

Coding the Working Storage Section

The first statement in the Working Storage Section *must* be:

```
$ Include Huron;
```

You then define the table or tables for your use. If table, parameter, or field names are invalid for COBOL, you must also define valid COBOL names in these statements. The following example defines the TDS table EMPLOYEE and re-assigns a valid COBOL name (mgrno) to a TIBCO Object Service Broker field (mgr#):

```
$ Define Table Employee = #ED_EMPLOYEES
$   Field mgrno = MGR#,
```

Refer to [Coding Considerations on page 346](#) for more information about assigning COBOL names to TIBCO Object Service Broker objects.

Coding the Procedure Division

In the Procedure Division, your first statement *must* be:

```
$ Entry;
```

Coding the Action Statements

The action statements must be at the end of the program, for example:

```
$ Forall Employee(USER-ID-77, LOCATION-77)
$   where mgrno = 84021 :
$       DISPLAY EMPLOYEE-LNAME.
$ end;
```

Coding Considerations

When coding your access statements, you must take into account the differences between COBOL and the TIBCO Object Service Broker rules language. For example:

- Valid TIBCO Object Service Broker names could be invalid in COBOL. For example, *table.field* names are invalid in COBOL.
- Error messages and codes are returned for TIBCO Object Service Broker access.
- The syntax of operators and expressions differ.
- The use of action statements differs.

Naming Differences Between TIBCO Object Service Broker and COBOL

The preprocessor creates COBOL names for each field by combining the table and field names with a hyphen. COBOL does not allow special symbols in table, field, and parameter names.

TIBCO Object Service Broker table and field names used in conjunction with a joining period (.) can have a maximum of 33 characters, but COBOL names can have a maximum of only 30 characters. The 30 character limit in COBOL is a result of the way that the preprocessor assigns COBOL names to TIBCO Object Service Broker fields (the TIBCO Object Service Broker identifier is two names and a separator while the COBOL identifier is a single name).

How to Rename TIBCO Object Service Broker Names to Valid COBOL Names

Use the Define Table statement in the Working Storage Section of your program to rename TIBCO Object Service Broker tables, fields, and parameters to valid COBOL names. You must assign valid names to TIBCO Object Service Broker invalid table, field, and parameter names in your table regardless of whether you refer to them or not.

You can assign alternate COBOL names for TIBCO Object Service Broker tables, fields, or parameters even though they are valid in COBOL. For example:

```
$ Define Table Employee = #ED_EMPLOYEES
$   Parameter USER-ID = USERID,
$       LOCATION-PARM = LOCATION,
$   Field mgrno = MGR#,
$   zipcode = P_CODE;
```

Coding Conventions



- The syntax of the reassigned names is:
`COBOL-name = OSB_name`
`COBOL-name = OSB_name`
- Unless the next declaration starts with a keyword such as `PARAMETER` or `FIELD`, each declaration must end with a comma (,).
- The statement ends with a semicolon (;).

Modifying the Join Character for Table.Field Names

Unlike the rules language, which uses a period (.) as the joining character to identify a table and field specification, TIBCO Object Service Broker access statements use a hyphen (-) to make them usable by COBOL. For example:

`Employee.lname`

is valid in the rules language, and

`Employee-lname`

is valid in a TIBCO Object Service Broker access statement.

Use the COBOL form to identify table and field specifications in TIBCO Object Service Broker access statements as well as in COBOL statements. For example, to specify this table and field in the Procedure Division, type:

`DISPLAY EMPLOYEE-LNAME.`

Checking TIBCO Object Service Broker Runtime Errors

You can check for TIBCO Object Service Broker access errors in the following variables:

HLL-RETURN-CODE	Contains 0 if the statement was successful, or 4 if the end of the table was encountered. An end-of-table is encountered when the FORALL action statement cannot return any more occurrences that satisfy the selection criteria. Any other code indicates an error. Refer to the entry HLL-RETURN-MESSAGE for more information.
HLL-RETURN-EXCEPTION	Contains the name of the exception that occurred.
HLL-RETURN-MESSAGE	Contains any generated TIBCO Object Service Broker message.

Coding Operators and Expressions

In COBOL, blanks must be inserted between an operator or expression and a variable. In TIBCO Object Service Broker, blanks are optional. The following example shows a valid selection in the access statement:

```
$ Forall Employee(USER-ID-77, LOCATION-77)
$   where mgrno = 84021 :
```

An expression must be a valid COBOL expression. Since a parameter expression is considered a literal expression in COBOL, you cannot use a parameter with a non-numeric literal with a length of zero. An expression like:

```
$GET employee(' ') where salary> 800
```

is *not valid*. The following example shows a valid GET statement that you can code in your access statement:

```
$GET employee where salary > 400
```

Embedding TIBCO Object Service Broker Action Statements

You can embed the following TIBCO Object Service Broker action statements in your COBOL program:

- GET

- INSERT
- DELETE
- REPLACE
- FORALL
- COMMIT
- ROLLBACK

In addition, the GETFIRST and GETNEXT statements are also supported. GETFIRST and GETNEXT work together to retrieve occurrences from a table within a COBOL looping structure.

GETFIRST Statement

The GETFIRST statement retrieves the first occurrence of a set of occurrences that meet the selection criteria, if any. The syntax is:

```
GETFIRST tablespec [ WHERE selection ]      [ ORDERED [ ASCENDING /  
DESCENDING-] field ] ;
```

where:

<i>tablespec</i>	Name of the table or table instance from which the occurrences are to be retrieved.
<i>selection</i>	Optional selection criteria to choose a set of occurrences.
<i>field</i>	Optional field whose values determine the order of the occurrences. The table can be ordered according to more than one field if you join the ORDERED clauses with the logical operator AND (&).

For example:

```
GETFIRST #ED_EMPLOYEES WHERE MGRNO = 79912 ORDERED ASCENDING LNAME;
```

This puts the occurrences in the order of ascending values in the **LNAME** field, and retrieves the first occurrence where the **MGRNO** field of the **#ED_EMPLOYEES** table is 79912. Other occurrences where the **MGRNO** field is 79912 are held in the same order for the GETNEXT statement.

GETNEXT Statement

The GETNEXT statement retrieves subsequent occurrences of a set of occurrences specified by the GETFIRST statement. The syntax is:

```
GETNEXT table;
```

table is the name of the table specified in the GETFIRST statement. You must include parameter values if the table is parameterized. An example of a valid GETNEXT statement is:

```
GETNEXT #ED_EMPLOYEES ;
```

If this statement appears after the example for GETFIRST, it retrieves the next occurrence from the #ED_EMPLOYEES table where the **MGRNO** field value is 79912.

See Also *TIBCO Object Service Broker Programming in Rules* about the rules language and rules processing behavior.

Chapter 22 **Coding SQL Access Statements**

This chapter describes how to code SQL access statements.

Topics

- [Writing a COBOL Program with Embedded SQL Statements, page 352](#)
- [Coding SQL Access Statements, page 356](#)
- [Coding Considerations, page 358](#)
- [Error Checking and Handling, page 361](#)
- [Statements Supported by the SQL Preprocessor, page 363](#)

Writing a COBOL Program with Embedded SQL Statements

TIBCO Object Service Broker tables can be accessed from your COBOL programs if you have coded SQL statements within them. These statements are converted to valid COBOL statements by the TIBCO Object Service Broker preprocessor. The COBOL programs participate with TIBCO Object Service Broker as a client as opposed to a server as outlined in this manual.

Sample COBOL Program

The following sample COBOL program contains embedded SQL statements. The following conventions are used:

- Statements in uppercase letters are COBOL statements.
- Statements in mixed case letters are SQL access statements.
- Statements beginning with an asterisk (*) are comments.



Parameter values must be typed in UPPERCASE.

```
CBL MAP,RENT,NOSEQUENCE,TEST(SYM),THREAD,VBREF,OFFSET
IDENTIFICATION DIVISION.
```

```
PROGRAM-ID. COBSQL RECURSIVE.
AUTHOR. JANE SMITH.
INSTALLATION. TIBCO Software Inc.
DATE-COMPILED.
```

```
*****
*
* Demonstration of HLI Preprocessor                               *
* A COBOL program with SQL statements to access Object Service  *
* Broker tables a in TSO or batch Object Service Broker session.*
*
* To execute this Cobol program, you must make an entry in the *
* Object Service Broker ROUTINES table such as:                  *
*   NAME           :COBSQL                                       *
*   LANGUAGE        :LEPERSIST                                    *
*   FUNCTION        :N                                           *
*   LOADNAME        :COBSQL                                       *
*
*
* Invoke the program via a RULE call:                             *
*   CALL 'COBSQL';                                              *
*
* The program uses the table #ED_EMPLOYEES as its data source.  *
*
```

```

*****
ENVIRONMENT DIVISION.
*
DATA DIVISION.
*
WORKING-STORAGE SECTION.
* set the NODENAME for table LOCATION parameter
  77 LOCATION77          PIC X(16) VALUE 'A'.
*
* Variables in this section remain in their last state for
* every invocation of this subroutine within a run-unit /
* enclave.
*
* Include Statement.
  Exec Sql
    Include Sqlca
  End-Exec
* Define table and fields of EMPLOYEE
  Exec Sql
    Define Employee Table = #ED_EMPLOYEES
      Parameter USER-ID = USERID,
                  LOCATION77 = LOCATION
      Field mgrno = MGR#,
             state-prov = PROV,
             zipcode = P_CODE
  End-Exec
  Exec Sql
    Declare Cursa Cursor For
      Select EMPNO, LNAME, MGR#
      From #ED_EMPLOYEES('EDUC',LOCATION77) where
      MGR#= 84021
  End-Exec
* Data areas to receive data from Object Service Broker
  01 COB-EMPNO PIC 9(7).
  01 LAST-NAME PIC X(22).
  01 MANAGER PIC 9(7).
*
LOCAL-STORAGE SECTION.
*
* Variables in this section are initialized at each invocation.
*
LINKAGE SECTION.
*
* Variables in this section are passed from/to caller on each
* invocation. Use ARGUMENTS table to define parameters to pass
* and the "PROCEDURE DIVISION USING parm1, parm2" etc to pass
* entries to the external routine as necessary. Each parameter
* should be a LINKAGE SECTION 01 entry.
*
PROCEDURE DIVISION.
MAINLINE-CODE SECTION.
  DISPLAY 'ENTERED COBSQL COBOL PROGRAM'.
* Open cursor
  DISPLAY 'ABOUT TO OPEN CURSOR'.
  Exec Sql
    Open Cursa
  End-Exec

```

```

        DISPLAY 'CURSOR OPENED'.
* Check for error on cursor open
        DISPLAY 'SQLCODE=', SQLCODE.
* Put Object Service Broker values in data areas
Exec Sql
    Fetch Cursa into :cob-EMPNO, :last-name, :manager
End-Exec
* Check for error on fetch
        DISPLAY 'SQLCODE=', SQLCODE.
* Display contents of two of the data areas
        DISPLAY 'RESULT OF FETCH ', COB-EMPNO, ' ', LAST-NAME.
* Close cursor
Exec Sql
    Close Cursa
End-Exec
* Check for error on cursor close
        DISPLAY 'SQLCODE=', SQLCODE.
NORMAL-EXIT-HERE.
        DISPLAY 'ABOUT TO EXIT COBSQL COBOL PROGRAM'.
        GOBACK.

```

Sample TIBCO Object Service Broker Table Definition

The following definition is of the table EMPLOYEE. This is the TIBCO Object Service Broker table used by the COBOL program with embedded SQL statements. (Only the first 80 columns are shown.)

Partial Definition of the EMPLOYEE Table

COMMAND==>													TABLE DEFINITION												
Table: #ED_EMPLOYEES													Type: TDS				Unit: EDUC				IDgen: N				
Parameter Name					Typ	Syn	Len	Dec	Class				'	Event Rule				Typ	Acc						
-----					-	-	----	--	-				'	-----				-	-						
_ USERID					I	C	16	0	D				'												
_ LOCATION					I	C	16	0	L				'												
Field Name					Typ	Syn	Len	Dec	Key	Ord	Rqd	Default	Reference												
-----					-	-	----	--	-	-	-	-----	-----												
_ EMPNO					I	P	3	0	P																
_ LNAME					S	C	22	0																	
_ POSITION					S	C	14	0																	
_ MGR#					I	P	3	0					MANAGER												
_ DEPTNO					I	B	2	0																	
_ SALARY					Q	P	3	2																	
_ HIREDATE					D	B	4	0																	
_ ADDRESS					S	V	38	0																	
_ CITY					S	C	20	0																	
_ PROV					S	C	3	0																	
_ P_CODE					S	C	7	0																	
_																									
_																									
PFKEYS: 3=END 12=CANCEL 22=DELETE 13=PRINT 14=FIELDS 21=DATA 2=DOC																									

Coding SQL Access Statements

Code the SQL access statements within the Working Storage Section and Procedure Division of your COBOL program. All SQL statements must be preceded by:

```
Exec Sql
```

and followed by:

```
End-Exec
```

Initial Statement

The first SQL statement in the Working Storage Section must be one of:

```
Exec Sql
  Include Sqlca
End-Exec
```

or

```
Exec Sql Begin Declare Section End-Exec
Exec Sql
  01 Sqlstate Pic x(5).
  01 Sqlcode Pic 59 Comp.End-Exec
Exec Sql End Declare Section End-Exec.
```

The *Sqlcode* declaration can be used in addition to, or in place of, the *Sqlstate* declaration. This is the method of error handling and error reporting through the SQL Communications Area (SQLCA), and the status variables *Sqlstate* and *Sqlcode*.

Defining Valid Names

If table, parameter, or field names are invalid for COBOL, define valid names. This example defines the TDS table #ED_EMPLOYEES and assigns a valid COBOL name (MGRNO) to a TIBCO Object Service Broker field (MGR#):

```
Exec Sql
Define Employee Table = #ED_EMPLOYEES
  Field mgrno = MGR#,
```

Refer to [Coding Considerations on page 358](#) for information about assigning COBOL names to TIBCO Object Service Broker objects.

Specifying Selection

Declare cursors to make selections from tables. Use standard SQL syntax and TIBCO Object Service Broker names:

```
Exec Sql
  Declare Cursa Cursor For
    Select EMPNO, LNAME, MGR#
    From #ED_EMPLOYEES('EDUC',LOCATION77) where
    MGR#= 84021
End-Exec
```

Specifying Data Areas

Specify data areas to receive data from TIBCO Object Service Broker:

```
01 COB-EMPNO PIC 9(7).
01 LAST-NAME PIC X(22).
01 MANAGER PIC 9(7).
```

Refer to [Syntax Mapping on page 360](#) for conversions from TIBCO Object Service Broker syntax to COBOL declarations.

Coding the Remaining SQL Statements

Code the remaining SQL statements as usual, such as FETCH CURSOR:

```
Exec Sql
  Fetch Cursa into :cob-EMPNO, :last-name, :manager
End-Exec
```



Close each cursor before using another cursor on the same table. You can use CLOSE CURSOR to close specific cursors. All cursors are closed by COMMIT and ROLLBACK statements.

Coding Considerations

Differences to Consider

Since you are accessing a TIBCO Object Service Broker database you must take into account differences between COBOL and TIBCO Object Service Broker. For example:

- Valid TIBCO Object Service Broker names could be invalid in COBOL; for example, TIBCO Object Service Broker *table.field* names are not valid in COBOL.
- Error checking has additional features for TIBCO Object Service Broker access.
- The syntax of operators and expressions differs.
- The supported SQL statements are a subset of all SQL statements.

Assigning Valid Names

Unlike TIBCO Object Service Broker, COBOL does not allow special symbols in table, field, and parameter names. TIBCO Object Service Broker table and field names used in conjunction with a joining period (.) can have a maximum of 33 characters; COBOL names can have a maximum of only 30 characters.

Rename TIBCO Object Service Broker tables, fields, and parameters to valid COBOL names using the Define Table statement in the Working Storage Section of your program.

Underscores (_) in TIBCO Object Service Broker table, field, or parameter names are automatically replaced with hyphens (-) if you do not explicitly rename them. You must assign valid names to names that contain the characters "\$", "@", or "#", regardless of whether you refer to them or not. You do not have to rename those that are already valid or converted, but you can if you want.

Renaming Fields Example

```
Exec Sql
Define Employee Table = #ED_EMPLOYEES
  Parameter USER-ID = USERID,
             LOCATION77 = LOCATION
  Field mgrno = MGR#,
        state-prov = PROV,
        zipcode = P_CODE
End-Exec
```


The field **MGR#** is renamed because the number sign (#) is an invalid COBOL character. Fields **STATE_PROV** and **ZP_CODE** did not have to be explicitly renamed. If they were left out of the Define Table statement, their COBOL names would be **STATE-PROV** and **ZP-CODE**.

Use the TIBCO Object Service Broker names in SQL statements and use COBOL names in COBOL statements.

The Form of a Statement



- The syntax of the re-assigned name is:
COBOL-name = OSB_name
- Unless the next declaration starts with a keyword, such as **PARAMETER** or **FIELD**, each declaration must end with a comma.
- The statement ends with no punctuation, and is followed by **END-EXEC**.

The Joining Character

Unlike SQL, which uses a period (.) as the joining character to identify a table and field specification, COBOL requires a (-) hyphen to make them usable. For example:

`Employee.lname`

is valid in SQL, and

`Employee-lname`

is valid in a COBOL statement. Use the SQL format in SQL statements and use the COBOL format in COBOL statements. An example of specifying a table and field in a COBOL statement in the Procedure Division is:

```
DISPLAY EMPLOYEE-LNAME
```

Coding Operators and Expressions

You can use either SQL or TIBCO Object Service Broker relational operators. All data conversions and calculations are done by COBOL.

An expression must be a valid COBOL expression. Since a parameter expression is considered a literal expression in COBOL, you cannot use a parameter with a non-numeric literal with a length of zero. An expression like:

```
Select EMPNO from EMPLOYEE('') where salary> 400
```

would be considered *invalid*. This is an example of a **SELECT** statement that you can code in your access statement:

```
Select EMPNO, LNAME, MGR#  
From #ED_EMPLOYEES('EDUC',LOCATION77) where  
MGR#= 84021
```

Syntax Mapping

For information on how TIBCO Object Service Broker syntax is mapped to COBOL syntax, refer to [Syntax Mapping on page 140](#).

See Also *TIBCO Object Service Broker Programming in Rules* about the rules language, rules processing behavior, and TIBCO Object Service Broker syntax.

Error Checking and Handling

Error Checking

After every SQL statement, check for errors. TIBCO Object Service Broker returns error information in the following variables:

<i>Sqlcode</i>	0 if the statement completed successfully, 100 if it reached the end of the table, or -1 if an error occurred.
<i>Sqlstate</i>	A string of five characters, containing error codes.
HRN-RETURN-EXCEPTION	The name of the TIBCO Object Service Broker exception that occurred.
HRN-RETURN-MESSAGE	Any TIBCO Object Service Broker message that was generated.

If you are using SQLCA, the variable *SQLERRMC* contains either an SQL message if an SQL error occurred or a TIBCO Object Service Broker error message.

Error Handling Status Variables

If SQLCA is present, it indicates that the *Sqlcode* variable is also present and error information must be relayed through the provided variables.

The *Sqlstate* status variable can be used in place of, or in addition to, the *Sqlcode* and SQLCA. It is a string of five characters that relays error codes. The following table shows the possible error codes and their explanations:

Sqlstate	Sqlcode	Explanation
00000	0	Successful completion.
02000	-1	No data (TABLEEND).
21000	-1	Cardinality violation (more than one item retrieved).
24000	-1	Invalid Cursor state.
42000	100	Syntax error/access violation.

You must ensure that you use the correct level number and picture clause: the preprocessor scans the host variables in the Declare section to see if *Sqlstate* is mentioned. If so, it updates all successes/errors through it, in addition to *Sqlcode* (if it was also mentioned in an Include SQLCA statement or on its own).

The *Sqlcode* status variable can be used on its own or together with *Sqlstate*. If SQLCA is used, there is no need to declare *Sqlcode*. If it is declared with SQLCA, the COBOL compiler can pick up two declarations and report it as an error at compile time. If you use *Sqlcode*, you must ensure that the proper level number and picture clause are used.

Any of these three methods can be used to get error information but the default assumption is that *Sqlcode* was properly declared. If this is not true (that is, *Sqlcode* was not declared), the compiler detects an undefined variable and reports it to you at compile time.

Statements Supported by the SQL Preprocessor

The following subset of SQL statements is supported by the SQL preprocessor. All these statements are written in the usual format. After each SQL statement, you should check for errors. Refer to [Error Checking on page 361](#) for more information.

SQL Statements

Statement	Usage
CLOSE CURSOR	You must close all the cursors before the end of the program. COMMIT and ROLLBACK also close all cursors.
COMMIT	Commits changes and closes all cursors.
DECLARE CURSOR	The DECLARE CURSOR statement accesses data from multiple tables or from a single table. Operation on a join type cursor (a cursor that accesses multiple tables) is limited to a fetch. The FOR UPDATE OF and ORDER BY clauses are not allowed on join type cursors.
DELETE	Delete all table occurrences satisfying the search condition. If no search condition is specified, all table occurrences are deleted. Both positioned delete (using WHERE CURRENT OF cursor-name) and searched delete are supported.
FETCH	You can have only one cursor open on a given table.
INSERT	Insert occurrence values into a table. The values are either those of a derived table of a query-specification or values specified directly: If the statement contains a query-specification, one row of values is inserted into the base table for each of the rows of the searched table that satisfy the search condition of the query-specification. If no search condition is stated, all the values of all the rows of the table in the query-specification are copied into the base table. The insertion is done in the order specified (that is, the value of the <i>n</i> th column of a row in the searched table, or the <i>n</i> th insert-value in the VALUES list, is assigned to the <i>n</i> th column of the base table).
OPEN CURSOR	All values for selection are determined at this point. WHENEVER is supported.

Statement	Usage
ROLLBACK	Rolls back changes and closes all cursors.
SELECT	Assign values from the specified table to host variables. The SELECT statement is supported only within the DECLARE CURSOR statement. Refer to Supported Keywords and Clauses for the SELECT Statement on page 365 for the keywords and clauses that are supported for the SELECT statement.
UPDATE	<p>This statement updates zero or more occurrences of the specified table. It behaves as follows:</p> <p>Both positioned update (which uses WHERE CURRENT OF cursor-name) and searched update are supported.</p> <p>If the WHERE clause is specified, only those occurrences satisfying the conditions are updated, otherwise all occurrences of the specified table are updated.</p> <p>Any expressions are evaluated in COBOL.</p> <p>Parameter information is taken from the CURSOR, not the table specification in UPDATE.</p> <p>If the SET clause changes the primary key, TIBCO Object Service Broker ignores the cursor. If the primary key exists, that occurrence is updated. If the primary key does not exist, the REPLACEFAIL exception is raised.</p>
WHENEVER	<p>Used in the Procedure Division of the COBOL program to enable or disable subsequent SQL statements from transferring control if they produce exceptions. It behaves as follows:</p> <p>The SQLERROR clause traps all errors except TABLEEND, which is trapped by the NOT FOUND clause.</p> <p>The WHENEVER statement affects those SQL statements that follow it.</p> <p>The keyword GOTO indicates the applicable SQL statements that follow should transfer control to the specified host-label.</p> <p>The keyword CONTINUE specifies no transfer of control in the SQL statements that follow it: this is typically used to cancel the redirection of a previous WHENEVER statement.</p>

Supported Keywords and Clauses for the SELECT Statement

The keywords and clauses supported by the SELECT statement are listed in the following table:

Keyword or Clause	Usage
ALL	DISTINCT is not supported.
FROM clause	The table specifications in the FROM clause can have parameters.
INTO clause	The specific host variables to retrieve values into.
ORDERED BY clause	<p>You can specify ASC (ascending order) or DESC (descending order), and you can refer to the fields by number.</p> <p>If the SELECT statement cannot find an occurrence that meets the specified conditions, it raises a TABLEEND exception and sets the applicable status variables with the appropriate codes. If you used the WHENEVER statement, the exception is trapped in the COBOL routine associated with the exception.</p> <p>If the SELECT statement finds an occurrence, it checks to see if there are any more occurrences that can be retrieved, and if so, it reports an error (<i>Sqlstate</i> = 21000). This is the desired behavior according to the various SQL standards. If there is only one occurrence meeting the conditions, the retrieved values are assigned to the corresponding host variables.</p>
WHERE clause	All search conditions are supported. You can use either SQL or TIBCO Object Service Broker relational operators. BETWEEN, AND, OR, and NOT are supported. NOT BETWEEN is not supported. With LIKE, you can use SQL wildcards (%) and (_) or TIBCO Object Service Broker wildcards (*) and (?).

Chapter 23 **Processing COBOL Programs**

This chapter describes how to process COBOL programs.

Topics

- [Preprocessing the Access Statements, page 368](#)
- [Preparing the Program, page 370](#)
- [Running the Program, page 372](#)

Preprocessing the Access Statements

Use the [HLIPREPROCESSOR](#) tool to preprocess your COBOL programs with embedded TIBCO Object Service Broker or SQL access statements. You can run this tool from the workbench. The output is a preprocessed COBOL program that can be compiled and linked. Alternatively, you can call it from JCL outside of TIBCO Object Service Broker. The JCL can preprocess, compile, and link in one job. Sample JCL is provided in members JCLOSTMT and JCLSQL of the JCL data set.

Usage of HLIPREPROCESSOR

The syntax for [HLIPREPROCESSOR](#) is as follows:

```
CALL HLIPREPROCESSOR(hostlang, imbedlang, infile, outfile, listfile, options);
```

where:

<i>hostlang</i>	Name of a programming language; use the string COBOL
<i>imbedlang</i>	The kind of statements that are embedded in the COBOL program; use the string HURON or SQL
<i>infile</i>	Name of the partitioned data set containing the COBOL program to be processed
<i>outfile</i>	Name of the allocated, partitioned data set to contain the processed COBOL program, which can be passed to the COBOL compiler
<i>listfile</i>	Name of the allocated data set to contain the output listing. This is optional if the COBOL program contains TIBCO Object Service Broker statements, and not applicable if it contains SQL statements.

options

A combination of the following strings: LIST or NOLIST and ERRORSTOP or NOERRORSTOP. This argument *must* be null if you have SQL statements in the COBOL program.

- LIST – Produce a listing.
 - NOLIST – Do not produce a listing.
 - ERRORSTOP – If an error is detected, stop processing and raise the exception STOP_AT_ERROR. If HLIPREPROCESSOR is executed from JCL, the exception STOP_AT_ERROR raises a completion code that prevents the next job step from executing. If an error is detected, the exception can stop processing before the compile and link-edit steps.
 - NOERRORSTOP – If an error is detected, do not stop processing and an exception is not raised.
-

Preparing the Program

Steps Required

Complete the following tasks to preprocess the program before executing it:

- 1. [Call HLIPREPROCESSOR with the appropriate arguments, page 370](#)
- 2. [Compile and link the program, page 370](#)
- 3. [Place the executable code in a load library, page 371](#)
- 4. [Identify the program to TIBCO Object Service Broker, page 371](#)

Task A Call HLIPREPROCESSOR with the appropriate arguments

The following options are available to you to run the [HLIPREPROCESSOR](#) tool:

- Use the workbench option EX execute rule, for example:
Ex execute rule ==> HLIPREPROCESSOR<Enter>
A screen appears where you enter values for its arguments.
- Call [HLIPREPROCESSOR](#) from within a rule, for example:

```
HLI_TEST;  
-----  
-  
CALL HLIPREPROCESSOR('COBOL', 'HURON',                                | 1  
'HR01.COBOL.SRCIN(COBOST1)', 'HR01.COBOL.SRCOUT(COBOUT1)',          |  
'HLL.HLLLIST', 'ERRORSTOP');                                       |  
-----
```

- Call [HLIPREPROCESSOR](#) from within JCL using the `RULE=rulename` statement, for example:
`RULE=HLIPREPROCESSOR('COBOL','SQL','HUR01.COBOL.SRCIN(COBSQL1)',
'HUR01.COBOL.SRCOUT(COBOST1)',',',',');`
Refer to the members JCLOSTMT and JCLSQL in the JCL data set for sample `RULE=HLIPREPROCESSOR` statements coded into JCL.

Task B Compile and link the program

Compile and link the program. Refer to the members JCLOSTMT and JCLSQL in the JCL data set for sample JCL to assist you in this procedure.

Task C Place the executable code in a load library

Place the compiled and linked code load module into a library concatenated to the DD statement HRNEXTR.

Task D Identify the program to TIBCO Object Service Broker

You must identify the program and its load module name to TIBCO Object Service Broker through an entry in the ROUTINES table. If the program has arguments, specify these in a table instance of the ARGUMENTS table. You must have adequate security clearance to insert data into these tables before editing them.

Refer to [Identifying Your External Routine to TIBCO Object Service Broker on page 152](#) for information about the ROUTINES and ARGUMENTS table.

Running the Program

Steps Required

After compiling and link-editing your preprocessed code, placing it in the load library, and identifying the program to TIBCO Object Service Broker, complete the following tasks to execute the COBOL program:

- 1. [Create a rule to call the program, page 372](#)
- 2. [Call the COBOL program, page 372](#)

Task A Create a rule to call the program

Using the ED edit rule option from the workbench, create a rule to call your program. For example, create the DEMO2 rule, to call the COBOL program named COBSQL:

```
DEMO2 ;
-----
-----+-----
CALL COBSQL;                               | 1
-----
```

Task B Call the COBOL program

When the program is in an executable code load library, and you have made the appropriate entries in the ROUTINES and ARGUMENTS tables, you can:

- Call the COBOL program from a TIBCO Object Service Broker rule, for example:
CALL COBSQL;
- Run it using S6BBATCH. Provide the name of the rule to the RULE=*rulename* parameter.
- Run it from S6BTSO. The data set containing the external routine must be concatenated to the HRNEXTR DD name in the calling CLIST.

Refer to [Chapter 4, TIBCO Object Service Broker Sessions Under z/OS Batch, on page 31](#) for information about S6BBATCH, and to [Chapter 5, TIBCO Object Service Broker Sessions Under TSO, on page 41](#) for information about S6BTSO.

See Also *TIBCO Object Service Broker Programming in Rules* about writing rules.

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, page 374](#)

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 the OSLI member of the H data set.

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 S6BDRAPI 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	PSWDINVALID	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	CORRRDATA	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	Rule call failed.
3092	UNDEFCEPAGE	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 rule 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

Symbols

- (hyphen), as join character [359, 347, 358](#)
- _ (underscore), as join character [358](#)
- . (period) as join character [347, 358](#)
- @IMSDCTRXOUT table [117](#)
- @IMSDCTRXS table, description [117](#)
- @SESSION tool
 - and CICS clients [88](#)
 - and the Call Level Interface [251](#)
- \$GETENVCOMMAREA tool
 - and CICS clients [67](#)
 - and the Call Level Interface [205](#)
 - example for CICS client [86](#)
 - passing data in, Call Level Interface [249](#)
 - to retrieve data from a started session [28](#)
 - to retrieve message input segments for IMS TM clients [102](#)
- \$GETOPT tool
 - to retrieve IMS TM message information [117](#)
 - to set IMS TM terminal options [105](#)
- \$SAVE macro, using for storage [161](#)
- \$SETENVCOMMAREA tool
 - and the Call Level Interface [205](#)
 - example for CICS client [89](#)
 - example for IMS TM clients [112](#)
 - passing data out, Call Level Interface [249](#)
- \$SETSESSIONEND tool
 - and batch clients [39](#)
 - and CICS clients [90](#)
 - and TSO clients [49](#)

Numerics

- 3270 terminal, logging in to session from [52](#)
- 3GL languages, supported by Call Level Interface [206](#)

A

- access errors, variables for [348](#)
- access statements. *See* TIBCO Object Service Broker
 - access statements; SQL access statements
- ACTION parameter, changing rules invocation [29](#)
- additional requirements for CICS Execution Environments [269](#)
- address space
 - as used by Call Level Interface [201, 206](#)
 - as used by TSO client [42](#)
 - client types that share [14, 14, 14](#)
- AMODE/RMODE attributes, as used by external routines [134](#)
- APF authorized data set, as used for the load module [55](#)
- architecture of TIBCO Object Service Broker, introduction to [2](#)
- ARGUMENTS table
 - and batch clients [40](#)
 - and CICS clients [91, 92](#)
 - and Native Execution Environment [57](#)
 - and TSO clients [50](#)
 - as used by external routines [152, 155](#)
 - as used by the Host Languages Interface [372](#)
- ASM data set, samples for environmental wait routine [262](#)
- available DDnames [37](#)

B

- batch applications, running [32](#)
- batch client [31–40](#)
 - See also* Call Level Interface; client programs; client styles; client types; Execution Environment parameters; external routines; session parameters

- ters
- and external routines 40
- controlling steps in JCL 39
- operation of TIBCO Object Service Broker client 33
- operation of user written client 35
- passing data to 39
- returning data from 39
- single-session type 12
- starting 32
- user written 34
- batch session
 - invoking 32
 - passing data to 39
 - returning data from 39

C

C program, and external routines

- as used with 146
- compatibility for 146
- sample program 148

calculate commarea size function

- for segment of certain size. *See* cliCommCreate1
- for segment of certain size. *See* cliCommSizeCalc1
- for segment of certain structure. *See* cliComm-SizeCalc
- total. *See* cliCommSize

calculate commarea size function. *See* cliCommSize

call a rule operation. *See* callrule

Call Level Interface 199–253

- calling a rule 205, 249
- calling module 207
- calling parameters 209
- calling parameters, summary 210
- calling sequence 212
- committing data changes 205, 247
- ending transaction 204, 247
- functional overview 203
- HRNHLLTM module parameters 209
 - summary of 210
- modifying transaction 204, 245
- obtaining startup parameters 234
- openness to TIBCO Object Service Broker data and

- resources 6
- operational characteristics 206
- purpose 200
- reason codes for 226
- return codes for 225
- rolling back data changes 205, 247
- samples provided in COBOL data set 220
- samples provided in MACRO data set 220
- specifying environmental wait routine 257
- starting Execution Environment 203, 234
- starting or ending stream 204, 243
- starting or ending transaction 243
- starting session 203, 239
- starting transaction 204
- stopping Execution Environment 203, 237
- stopping session 203, 241
- storage, obtaining 251
- supported 3GL languages 206
- supported connections 201
- supported functionality 200
- use of address space 201
- user client usage 202

call SDK (Java) method 314

callrule cliProc operation 283

CALLRULE function, description 249

CICS applications, running 60

CICS client 60–92

- See also* Call Level Interface; client programs; client styles; client types; Execution Environment parameters; external routines; session parameters
- available DDnames 69
- listing of available client programs 66
- multiple-session type 13
- non-seamless. *See* non-seamless client
- operation of a user written client 62
- operation of TIBCO Object Service Broker client 60
- seamless. *See* seamless client
- selecting client program 66
- selecting client program for SDK (C/C++) 269
- starting 60

CICS COMMAREA

- description of non-seamless 85
- description of seamless 86
- passing to TIBCO Object Service Broker CICS

- session 85
- retrieving in a rule 86
- usage by seamless TIBCO Object Service Broker
 - CICS client 86
 - usage by TIBCO Object Service Broker CICS clients 85
- CICS session
 - See also* session parameter values
 - calling external routines from 91
 - CICS functions to perform 90
 - external routines restrictions 91
 - passing COMMAREA to 85
 - starting 64
 - starting at the command line 70
 - starting with EXEC CICS LINK 77
 - starting with EXEC CICS LINK with Channel 79
 - starting with EXEC CICS START 70
 - starting with EXEC CICS START with Channel 73
 - starting with EXEC CICS XCTL 81
 - starting with EXEC CICS XCTL with Channel 82
 - terminating 64
- CICS session background task, program name 269
- CICS transaction, replacing with rules 65
- CICSVSAMSYNC Execution Environment
 - parameter 69
- cliCommCreate SDK (C/C++) function 294
- cliCommCreate1 SDK (C/C++) function 294
- cliCommDelete SDK (C/C++) function 295
- cliCommFormat SDK (C/C++) function 295
- cliCommFormat1 SDK (C/C++) function 296
- cliCommSegment SDK (C/C++) function 297
- cliCommSegments SDK (C/C++) function 297
- cliCommSegSize SDK (C/C++) function 298
- cliCommSize SDK (C/C++) function 293, 298
- cliCommSizeCalc SDK (C/C++) function 299
- cliCommSizeCalc1 SDK (C/C++) function 299
- client
 - conversational style, description 17
 - determining style for CICS and IMS TM 15
 - display style, description 16
 - external transaction style, description 16
 - invoking. *See* starting
 - model in TIBCO Object Service Broker 12
 - non-conversational style, description 17
 - non-display style, description 16
 - non-seamless style, description 15
 - seamless style, description 15
 - selecting CICS client program 66
 - selecting CICS client program for SDK (C/C++) 269
 - starting batch client 32
 - starting CICS client 60
 - starting IMS TM client 98
 - starting Native Execution Environment 52
 - starting TSO client 42
 - styles for CICS and IMS TM 15
 - types of 12, 12
- client as part of COBOL, PL/1, C, or assembler
 - application 13
- client programs
 - associating IMS transaction identifier 104
 - CICS client programs available 66
 - IMS TM client program input message
 - format 109–111
 - IMS TM client programs available 102
 - summary of programs available 17
 - use of exit routines with IMS TM client 119
- client services layer
 - explanation of 2
 - purpose of 2
- client style summary 17
- client types, description 12
- cliExecTran SDK (C/C++) function 289
- cliProc operations
 - callrule 283
 - getendmsg 287
 - resetss 287
 - sessactive 288
 - startss 280
 - starttr 282
 - stopss 286
 - stoptr 286

- cliProc SDK (C/C++) function [277](#)
- cliSetCodepage SDK (C/C++) function [291](#)
- CLIST
 - distributed with TIBCO Object Service Broker [48](#)
 - USER distributed with TIBCO Object Service Broker [48](#)
- CLOSE CURSOR statement
 - and SQL access statements [357](#), [363](#)
 - usage with SQL access statements [363](#)
- COBCAPI3 sample program [264](#)
- COBOL data set
 - samples for Call Level Interface [220](#)
 - samples for Host Languages Interface [341](#)
- COBOL names
 - coding conventions for renaming [347](#)
 - renaming TIBCO Object Service Broker names to COBOL names [346](#), [358](#)
- COBOL program, and external routines
 - as used with [138](#)
 - compatibility for [138](#)
 - example program [141](#)
- COBOL program, and Host Languages Interface
 - compiling and linking [370](#)
 - identifying to TIBCO Object Service Broker [371](#)
 - preprocessing steps for [370](#)
 - running after preprocessing [372](#)
- COBOL program, and SQL access statements
 - assigning valid names [358](#)
 - coding [356](#)
 - coding considerations [358](#)
 - coding operators and expressions [359](#)
 - defining valid names [356](#)
 - differences between TIBCO Object Service Broker and COBOL usages [358](#)
 - how to access TIBCO Object Service Broker data [340](#)
 - initial statements for [356](#)
 - sample program [352](#)
 - specifying data areas [357](#)
 - specifying selection [357](#)
 - writing [352](#)
- COBOL program, and TIBCO Object Service Broker
 - access statements
 - checking for runtime errors [348](#)
 - coding action statements [345](#)
 - coding considerations [346](#)
 - coding for [345](#)
 - coding operators and expressions [348](#)
 - coding Procedure Division [345](#)
 - coding Working Storage Section [345](#)
 - how to access TIBCO Object Service Broker data [340](#)
 - modifying table.field names [347](#)
 - naming differences between TIBCO Object Service Broker and COBOL [346](#)
 - sample with [342](#)
 - writing [342](#)
- COBOL program, sample source for TIBCO Object Service Broker and SQL access statements [341](#)
- code page support, and EMS interface [173](#)
- COMMAREA. *See* CICS COMMAREA
- commCreate SDK (Java) method [330](#)
- commFormat SDK (Java) method [330](#)
- COMMIT statement
 - and SQL access statements [357](#), [363](#)
 - and TIBCO Object Service Broker action statements [349](#)
- committing data changes, with Call Level Interface [205](#), [231](#), [247](#)
- commSegmentInd SDK (Java) method [331](#)
- commSegments SDK (Java) method [331](#)
- commSegSize SDK (Java) method [332](#)
- commSize SDK (Java) method [332](#)
- commSizeCalc SDK (Java) method [333](#)
- constants for SDK (C/C++) [273](#)
- constants for SDK (Java) [307](#)
- controlling steps in JCL for batch client [39](#)
- conversational client
 - description [17](#)
 - summary of program names [17](#)
- conversational interface [17](#)
- customer support [xxv](#)

D

data

- access to 6
- committing data changes with Call Level Interface 205, 247
- obtaining 28
- retrieving from a CICS COMMAREA 86
- rolling back data changes with Call Level Interface 205, 247
- stages to processing 8

data conversions, and COBOL 359

Data Object Broker, explanation of 3

data passing

- to batch client 39
- to Native Execution Environment 57
- to TSO client 49, 49

data returning

- from batch client 39
- from TSO client 49
- to CICS COMMAREA 88
- to Native Execution Environment 57

data store, explanation of 3

DB2, modifying access to 340

DDnames

- available for CICS client 69
- available for Native Execution Environment 55
- DFHRPL, description 69
- HRNEXTR, description 37, 55, 69
- HRNIN, description 37, 55, 69
- HRNLIB, description 37, 55
- HRNOUT, description 37
- HRNPRNT, description 37
- STEPLIB, description 37, 55, 69

DECLARE CURSOR statement

- and SQL access statements 363
- usage with SQL access statements 363

delete a commarea function. *See* cliCommDelete

DELETE statement

- and SQL access statements 363
- and TIBCO Object Service Broker action statements 349
- usage with SQL access statements 363

DFHRPL DDname, description 69

DISPLAY & TRANSFERCALL statement, using in a

display client 16

display client

- description 16
- ruled statements to use 16
- summary of program names 17

DISPLAY statement, using in a display client 16

drop a connection to a session operation. *See* resetss

E

EECONFIG, and session configuration 54, 234

EMS interface 168–194

- code page support 173
- configuring 173
- LE Enclave 173
- sample applications 175
- shareable tools 169
- supported functions 177

endMessage SDK (Java) method 317

ENDMSG tool

- and CICS COMMAREA 85
- location of the end message 87

environmental wait routine

- implementation 256
- sample programs 259, 261, 262
- specifying with Call Level Interface 257

error handling

- by external routines 133
- by MOM routines 172, 197
- by OTMA 129
- checking after SQL access statements 361
- checking for runtime errors in TIBCO Object Service Broker access statements 348
- recoverable errors 28

error reason codes, SDK (C/C++)

- 106 [282](#)
- 128 [287](#)
- 161 [293](#)
- 193 [281](#)
- 195 [281](#)
- 199 [279](#), [287](#)
- 3090 [284](#)
- 36 [281](#), [282](#), [284](#), [286](#), [287](#), [289](#)
- 96 [284](#)
- listing and explanation of [374](#)

error reason codes, SDK (Java)

- 106 [324](#)
- 193 [322](#)
- 3090 [315](#)
- 36 [315](#), [324](#), [325](#)
- 96 [315](#)

error trapping, and reducing session resources [28](#)errorReasonDescr SDK (Java) method [328](#)

EXEC CICS LINK

- assembler example [77](#), [79](#)
- starting CICS client with [77](#)

EXEC CICS RETURN [90](#)

EXEC CICS START

- assembler example [71](#)
- COBOL example [73](#)
- starting CICS client with [70](#)

EXEC CICS START TRANSID

- assembler example [74](#)
- COBOL example [76](#)

EXEC CICS XCTL

- assembler example [81](#), [82](#)
- starting CICS client with [81](#)

EXECLOCALSIZE parameter

- CICS recommendations for [87](#)

EXECSTACKSIZE parameter

- CICS recommendations for [87](#)

execTran SDK (Java) method [318](#)

execute a transaction function. *See* [cliExecTran](#)

Execution Environment

- batch client, sequential file for [37](#)
- determining type of [22](#), [23](#)
- explanation of [3](#)
- load modules [37](#)
- locating [235](#)
- obtaining startup parameters with Call Level Interface [234](#)
- preparing to start or locate with Call Level Interface [220](#)
- starting for CICS sessions [64](#)
- starting multiple sessions in batch [264](#)
- starting with Call Level Interface [203](#), [234](#), [235](#)
- stopping with Call Level Interface [203](#), [237](#)

Execution Environment parameters

- file containing [55](#)
- listed [25](#)

exit routine, as used with IMS TM client programs [119](#)EXLIB CLIST parameter [48](#)

external database servers

- and Native Execution Environment [52](#)
- explanation of [4](#)

external environment

- explanation of [2](#), [5](#)
- interaction with TIBCO Object Service Broker [8](#)
- supported types [2](#)

- external routines 132–157
 - and batch client 40
 - and Native Execution Environment 57
 - and TSO client 50
 - C program compatibility 146
 - calling from CICS client 91
 - CICS restrictions 91
 - CICS usage with OS linkage 92
 - cleanup of system service requests 133
 - COBOL program compatibility 138
 - error handling 133
 - example assembler program 136
 - example COBOL program 141
 - example PL/I program 144
 - identifying to TIBCO Object Service Broker 152
 - information available 134
 - load library 48
 - load module for user external routines 55
 - load modules for user written 37
 - openness from TIBCO Object Service Broker 6
 - PL/I program compatibility 143
 - processing by TIBCO Object Service Broker 132
 - sample C program 148
 - steps required to use 132
 - storage requirements 135
 - transaction level 133
 - use of AMODE/RMODE attributes 134
 - use of ARGUMENTS table 152, 155
 - use of load library 157
 - use of ROUTINES table 152
- external security
 - transaction security 16
 - user security 16
 - using TIBCO Object Service Broker ID as user ID 16
- external transaction client
 - description 16
 - summary of program names 17
- external transaction name, user ID required for 18
- external user client, summary of program names 17

F

- FETCH statement
 - and SQL access statements 363
 - usage with SQL access statements 363
- file
 - sequential for batch client 37
 - session parameter input 27
 - session parameter, for batch 36
 - session parameter, for Native Execution Environment 54
 - session parameter, for TSO 46
- FORALL statement, and TIBCO Object Service Broker
 - action statements 349
- format a commarea function
 - for multiple segments. *See* cliCommFormat
 - for one segment. *See* cliCommFormat1
 - for segment of certain structure. *See* cliCommCreate
- functional overview of Call Level Interface 203
- functions, of Call Level Interface
 - CALLRULE 249
 - STARTEE 234, 259
 - STARTSS 239, 261
 - STARTTR 243
 - STOPEE 237
 - STOPSS 241
 - STOPTR 247
- functions, supported for EMS 177

G

- Gateway
 - for Adabas and Native Execution Environment 52, 54
 - for Datacom and Native Execution Environment 52, 54
 - for IMS/DB and Native Execution Environment 52, 54
- GENBIN tool, and IMS TM 113
- GET statement, and TIBCO Object Service Broker
 - action statements 348
- getendmsg cliProc operation 287

GETFIRST statement
 and TIBCO Object Service Broker action
 statements 349
 example 349

GETNEXT statement
 and TIBCO Object Service Broker action
 statements 349
 example 349

H

HINT transaction, use of 64

HLIPREPROCESSOR tool
 examples to run 370
 usage to preprocess TIBCO Object Service Broker or
 SQL access statements 368

HLL-RETURN-CODE variable, for TIBCO Object Service Broker access statements 348

Host Languages Interface 340–372
 See also batch client; Call Level Interface; external routines; TSO client

HRNEXTR DDname
 description 55

HRNEXTR DDname, description 37, 47, 69

HRNHLLTM module
 and Call Level Interface 207
 required parameters 207
 valid parameters 209
 valid parameters, summary 210

HRNIN DDname
 description 37, 55, 69
 format for use 27
 restriction on use 54
 sample usage 38

HRNLIB DDname, description 37, 55

HRNOUT DDname
 description 37
 restriction on use 55, 69

HRNPRNT DDname
 description 37
 restriction on use 55, 69

HRNXD copybook, usage with Call Level Interface 257

HURN transaction
 explanation 70
 usage 70

HURON trancode, usage of 103

hyphen (-), as join character 347

I

IMS OTMA. *See* OTMA support

IMS TM applications, running 98

IMS TM client 98–119
 See also Call Level Interface; client programs; client styles; client types; Execution Environment parameters; external routines; session parameters
 and the Call Level Interface 98
 associating IMS transaction identifier 104
 bypassing the user ID and password 104
 establishing a session 98
 listing of available programs 101
 multiple-session type 13
 operation of a user written client 100
 operation of TIBCO Object Service Broker client 99
 use of exit routines 119

IMS TM client programs, summarized 102

IMS TM programs, replacing with rules 100

IMS TM session
 ensuring message queue and database consistency 117
 exit routine replacement 119
 invoking. *See* starting
 starting 103
 starting with supplied trancode 103
 using MFS to start 104

IMS TM terminals
 extended terminal support 105
 PF key changes 105
 signon exit 105
 starting a session from 103

IMS transaction identifier, associating with IMS TM client program 104

IMSSCREENATTRIBU parameter
 to set IMS TM terminal characteristics 105

inquire whether a session is active operation. *See* session active

INSERT statement

- and SQL access statements 363
- and TIBCO Object Service Broker action statements 349
- usage with SQL access statements 363

invoking rules

- as new transaction 29
- as part of logon transaction 29

isActive SDK (Java) method 320

J

JCL

- using to control steps in batch session 39
- using to invoke batch session 32

JCL data set

- samples for Host Languages Interface 341

JES2/JES3

- conditional processing 39
- starting batch session under 32

join character

- for COBOL 347
- for TIBCO Object Service Broker access statements 347

L

LE Enclave, and EMS 173

list of Execution Environment parameters 25

list of session parameters 26

LLCOPY_CSTR(listr, cstr) SDK (C/C++) function 300

LLCOPY_MEM(listr, prt, len) SDK (C/C++) function 300

LLDECLARE(name, len) SDK (C/C++) function 300

LLSETLEN(listr, len) SDK (C/C++) function 300

LLSTR(listr) SDK (C/C++) function 300, 300

load library

- as used by external routines 157
- specifying in a CLIST 48

load module

- and external routines 134
- for Execution Environment 37
- for user external routines 55
- for user written external routines 37
- linking executable COBOL code 371
- required to run Native Execution Environment 55

LOADLIB 47

LU2 (3270) terminal, logging in to session from 52

M

MACLIB data set, samples for the Call Level Interface 220

MAP tables

- accessing data with CICS client 86
- as used by Call Level Interface 201
- to access data with Call Level Interface 205
- to pass data to IMS TM session 106
- to return data from IMC/DC session 112
- to return data to batch client 40
- to return data to CICS client 88
- to return data to TSO client 49
- used to obtain input message segments 102
- where used 28

Message Formatting Services (MFS)

- and TIBCO Object Service Broker interaction 100
- used to start an IMS TM session 104

Message Oriented Middleware

- error handling 172, 197
- example rule 197
- usage notes 169, 196

Message Processing Region (MPR), and access to TIBCO Object Service Broker 98

MESSAGE_LOG tool, as used by Call Level Interface 222

MetaStor, explanation of 3

modifying, rules invocation 29

MOM. *See* Message Oriented Middleware

MQ Series, accessing 196

multiple-session client types 13

multiple-session Execution Environments 256–264

N

Native Execution Environment 52–57

See also Call Level Interface; client types; Execution Environment parameters; external routines; session parameters

and remote peer server 52

and session parameter values 54

and VTAM LU2 52

starting 52

Native Execution Environment client 13, 13

multiple-session type 13

network solicitor, returning to 57

NLS requirement 305

non-conversational client

summary of program names 17

non-conversational client, description 17

non-conversational interface 17

non-display client

description 16

summary of program names 17

non-seamless client

and IMS TM 102

description 15

summary of program names 17

usage of CICS clients 67

non-seamless COMMAREA, description 85

non-seamless session, executing rule at startup 28

O

OPEN CURSOR statement, usage with SQL access statements 363

OTMA support

definition of OTMA 126

error handling 129

example rules and tables 129

programming for 127

usage notes 129

P

Pagestore, purpose of 3

parameters, specifying for session 21

See also batch client; Call Level Interface; CICS client; Native Execution Environment; TSO client

passing COMMAREA, to TIBCO Object Service Broker CICS session 85

passing data

to Native Execution Environment 57

to TSO client 49

password, bypassing with IMS TM client 104

period (.) as join character 347, 358

PL/I

example program for external routines 144

program compatibility for external routines 143

Procedure Division, coding TIBCO Object Service Broker access statements in 345

processing data, stages to 8

Q

query number of segments in commarea function, total. *See* cliCommSegments

R

rc SDK (Java) method 329

readInt SDK (Java) method 333

readShort SDK (Java) method 334

reason codes, listing and explanation of 226

reason codes, SDK (C/C++) errors

- 106 [282](#)
- 128 [287](#)
- 161 [293](#)
- 193 [281](#)
- 195 [281](#)
- 199 [279](#), [287](#)
- 3090 [284](#)
- 36 [281](#), [282](#), [284](#), [286](#), [287](#), [289](#)
- 96 [284](#)
- listing and explanation of [374](#)

reason codes, SDK (Java) errors

- 106 [324](#)
- 193 [322](#)
- 3090 [315](#)
- 36 [315](#), [324](#), [325](#)
- 96 [315](#)

reasonCode SDK (Java) method [328](#)

recoverable errors, trapping [28](#)

reducing session resources [28](#)

remote peer server, and Native Execution Environment [52](#)

REPLACE statement, and TIBCO Object Service Broker action statements [349](#)

requirements [127](#)

NLS [305](#)

runtime [305](#)

reset SDK (Java) method [320](#)

resetss cliProc operation [287](#)

retrieve a rules end message operation. *See* getendmsg

retrieve segment size function. *See* cliCommSegSize

return codes

- evaluating [222](#)
- listing and explanation of [225](#)

return pointer to commarea function. *See* cliCommSegment

RETURN_CODE tool, as used by external routines [134](#)

returning data

- from batch session [39](#)
- from TSO client [49](#)
- to CICS COMMAREA [88](#)
- to Native Execution Environment [57](#)

ROLLBACK statement

- and SQL action statements [357](#)
- and TIBCO Object Service Broker action statements [349](#)

ROLLBACK statement, usage with SQL access statements [364](#)

rolling back data changes, with Call Level Interface [205](#), [231](#), [247](#)

routine, environmental wait. *See* environmental wait routine

ROUTINES table

- and batch clients [40](#)
- and CICS clients [91](#), [92](#)
- and Native Execution Environment [57](#)
- and TSO clients [50](#)
- as used by external routines [152](#)
- as used by Host Languages Interface [372](#)

RULE parameter

- and seamless clients [102](#)
- in non-seamless sessions [28](#)
- passing data to batch sessions [39](#)
- passing data to TSO sessions [49](#)

rules

- calling with Call Level Interface [205](#), [249](#)
- library search order specification in Call Level Interface [204](#), [204](#)
- replacing CICS transaction with [65](#)
- replacing IMS TM programs with [100](#)
- retrieving CICS COMMAREA with [86](#)

rules invocation

- as new transaction [29](#)
- as part of logon transaction [29](#)
- for COBOL program with access statements [372](#)

runtime requirement [305](#)

S

S6BBATCH program

- description [32](#)
- using to run a COBOL program with access statements [372](#)

S6BCALL tool [169](#)

S6BCSCL1 program [269](#)

- S6BCSCLI, CICS session background task
 - program [269](#)
- S6BCSSC1 program [66, 66](#)
- S6BCSSC2 program [66, 70, 74, 76, 77, 81, 82](#)
- S6BCSSN1 program [66](#)
- S6BCSSN2 program [66](#)
- S6BCSTC1 program [66](#)
- S6BCSTC2 program [66](#)
- S6BCSTN1 program [66](#)
- S6BCSTN2 program [66](#)
- S6BDCKRN program [101](#)
- S6BDCSGN signon exit [105](#)
- S6BDCUSX exit routine [119](#)
- S6BDR00 program [54](#)
- S6BDRCB0 module [36](#)
- S6BDRCN0 module [54](#)
- S6BDRCT0 module [46](#)
- S6BEWTIN program [262](#)
- S6BEWTSD sample program [262](#)
- S6BEWTSS sample program [263](#)
- S6BFUNCTION tool [169](#)
- S6BIMSC1 program
 - introduction [101](#)
- S6BIMSC1 program, input message format [109](#)
- S6BIMSC2 program
 - introduction [101](#)
- S6BIMSC2 program, input message format [109](#)
- S6BIMSN1 program
 - introduction [101](#)
- S6BIMSN1 program, input message format [110](#)
- S6BIMSN2 program
 - introduction [101](#)
- S6BIMSN2 program, input message format [111](#)
- S6BIMSxx program, description [102](#)
- S6BIMTC1 program
 - introduction [101](#)
- S6BIMTC1 program, input message format [109](#)
- S6BIMTC2 program
 - introduction [101](#)
- S6BIMTC2 program, input message format [109](#)
- S6BIMTN1 program
 - introduction [101](#)
- S6BIMTN1 program, input message format [110](#)
- S6BIMTN2 program
 - introduction [101](#)
- S6BIMTN2 program, input message format [111](#)
- S6BIMTxx program, description [102](#)
- S6BIMxCx program, description [102](#)
- S6BIMxNx program, description [102](#)
- S6BIMxx1 program, description [102](#)
- S6BIMxx2 program, description [102](#)
- S6B-RETURN-EXCEPTION variable
 - for SQL access statements [361](#)
 - for TIBCO Object Service Broker access
 - statements [348](#)
- S6B-RETURN-MESSAGE variable
 - for SQL access statements [361](#)
 - for TIBCO Object Service Broker access
 - statements [348](#)
- S6BSBDR module, used by IMS TM terminals [105](#)
- S6BTSO program
 - description [42](#)
 - using to run a COBOL program with access
 - statements [372](#)
- sample application
 - using SDK (C/C++) [301](#)
 - using SDK (Java) [336](#)
- sample applications, for EMS interface [175](#)
- sample JCL, for preprocessing COBOL programs [368](#)
- sample rules [127](#)
- SDK (C/C++)
 - defined [272](#)
 - error reason codes for [374](#)
 - how to use [272](#)
- SDK (C/C++) constants [273](#)
- SDK (C/C++) error reason codes
 - 106 [282](#)
 - 128 [287](#)
 - 161 [293](#)
 - 193 [281](#)
 - 195 [281](#)
 - 199 [279, 287](#)
 - 3090 [284](#)
 - 36 [281, 282, 284, 286, 287, 289](#)
 - 96 [284](#)
 - listing and explanation of [374](#)

SDK (C/C++) functions

- cliCommCreate [294](#)
- cliCommCreate1 [294](#)
- cliCommDelete [295](#)
- cliCommFormat [295](#)
- cliCommFormat1 [296](#)
- cliCommSegment [297](#)
- cliCommSegments [297](#)
- cliCommSegSize [298](#)
- cliCommSize [293, 298](#)
- cliCommSizeCalc [299](#)
- cliCommSizeCalc1 [299](#)
- cliExecTran [289](#)
- cliProc [277](#)
- cliSetCodepage [291](#)
- LLCOPY_CSTR(listr, cstr) [300](#)
- LLCOPY_MEM(listr, prt, len) [300](#)
- LLDECLARE(name, len) [300](#)
- LLSETLEN(listr, len) [300](#)
- LLSTR(listr) [300, 300](#)

SDK (C/C++), introduction [266](#)

SDK (Java)

- defined [304](#)
- how to use [306](#)

SDK (Java) constants [307](#)

SDK (Java) error reason codes

- 106 [324](#)
- 193 [322](#)
- 3090 [315](#)
- 36 [315, 324, 325](#)
- 96 [315](#)

SDK (Java) methods

- call [314](#)
- commCreate [330](#)
- commFormat [330](#)
- commSegmentInd [331](#)
- commSegments [331](#)
- commSegSize [332](#)
- commSize [332](#)
- commSizeCalc [333](#)
- endMessage [317](#)
- errorReasonDescr [328](#)
- execTran [318](#)
- isActive [320](#)
- rc [329](#)
- readInt [333](#)
- readShort [334](#)
- reasonCode [328](#)
- reset [320](#)
- Session [312](#)
- SessionException [327](#)
- shutdown [321](#)
- start [321](#)
- startTrans [323](#)
- stop [324](#)
- stopTrans [325](#)
- transNestLevel [325](#)
- userId [326](#)
- writeInt [334](#)
- writeShort [335](#)

seamless client

- and CICS [67](#)
- and IMS TM [102](#)
- CICS COMMAREA usage [86](#)
- description [15](#)
- summary of program names [17](#)

security

- external transaction [16](#)
- external user [16](#)

SELECT statement

- keywords for [365](#)
- usage with SQL access statements [364](#)

selecting client program

- for CICS [66](#)
- for IMS TM [101](#)
- for SDK (C/C++) within CICS [269](#)

- Service Gateway for CICS 60
- Service Gateway for IMS TM 98
- Service Gateway for WMQ 196
- sessactive cliProc operation 288
- session
 - defined 3
 - multiple-session types 13
 - reducing resources of 28
 - single-session types 12
 - standby for Call Level Interface 206
 - starting multiple-session Execution Environments in batch 256
 - starting with Call Level Interface 203, 239
 - stopping with Call Level Interface 203, 241
- session parameter input file 27
- session parameter values
 - evaluating for batch client 36
 - evaluating for CICS client 68
 - evaluating for Native Execution Environment 54
 - evaluating for TSO session 46
 - instream list for batch client 38
 - sequential file containing for batch client 37
 - setting for batch session 36
 - setting for CICS session 68, 68
 - setting for IMS TM session 103, 103
 - setting for Native Execution Environment 54
 - setting for TSO session 46
 - starting CICS client at command line 70
- session parameters
 - ACTION, changing rules invocation 29
 - listed 26
 - sequential file containing 55
- session resources, reducing 28
- Session SDK (Java) method 312
- session startup
 - CLIST syntax for TSO clients 47
 - default for Native Execution Environment 54
 - installation defaults for batch clients 36
 - installation defaults for TSO clients 46
 - limitations to startup string and input file 54
 - load library in CLIST 48
 - parameter input file for a Native Execution Environment 54
 - parameter input file for batch clients 36
 - parameter input file for TSO clients 46
 - sample CLIST for TSO client 47
 - startup string for batch clients 36
 - startup string for Native Execution Environment 54
 - startup string for TSO clients 46
 - TIBCO Object Service Broker defaults for batch clients 36
 - TIBCO Object Service Broker defaults for TSO clients 46
 - user profile for batch clients 36
 - user profile for TSO clients 46
- session termination 128
 - CICS functions to perform 90
 - handling in CICS client 87
- SESSIONENDACTION parameter and CICS 90
- SESSIONENDVALUE parameter and CICS 90
- SessionException SDK (Java) method 327
- set code page function
 - See also* cliSetCodepage
- shutdown SDK (Java) method 321
- SIGNON transaction, for IMS TM clients 104
- single-session clients, types of 12
- specifying
 - startup rule explicitly 15
- SQL access statements
 - and COBOL programs 340
 - assigning valid names 358
 - coding 356
 - coding operators and expressions 359
 - defining valid names 356
 - differences between TIBCO Object Service Broker and COBOL usages 358
 - initial statements for 356
 - sample program 352
 - specifying data areas 357
 - specifying selection 357
 - supported 363
 - writing COBOL program for 352
- Sqlcode variable, for SQL access statements 361
- Sqlstate variable, for SQL access statements 361

STANDBYNUM parameter
 as used by Call Level Interface 206

start a session operation. *See* startss

start a transaction operation. *See* starttr

start SDK (Java) method 321

STARTEE function
 description 234
 environment wait routine, summary of usage 236
 environmental wait routine 259

starting 98
 batch client 32
 CICS client 60
 IMS TM client 98
 Native Execution Environment client 52, 53
 TSO client 42
 VTAM LU2 (3270) terminal session 53

startss cliProc operation 280

STARTSS function
 description 239
 environmental wait routine 261
 environmental wait routine, summary of usage 240

starttr cliProc operation 282

STARTTR function, description 243

startTrans SDK (Java) method 323

startup rule, explicitly specifying 15

STEPLIB DDname, description 37, 55, 69

steps in JCL for batch client, controlling 39

stop a session operation. *See* stopss

stop a transaction operation. *See* stoptr

stop SDK (Java) method 324

STOPEE function, description 237

stopss cliProc operation 286

STOPSS function, description 241

stoptr cliProc operation 286

STOPTR function, description 247

stopTrans SDK (Java) method 325

storage
 assembler requirements and external routines 135
 obtaining with Call Level Interface 251

storage, using \$SAVE macro for 161

stream, starting or ending with Call Level Interface 204, 243

summary of client styles 17

supplied session defaults, TSO session 46

support, contacting xxv

synchronization under CICS 69

syntax mapping
 TIBCO Object Service Broker to C 147
 TIBCO Object Service Broker to COBOL 140, 360
 TIBCO Object Service Broker to PL/I 144

T

table definition, sample 344, 354

table.field names, modifying for TIBCO Object Service Broker access statements 347

tables
 @IMSDCTRXOUT 117
 @IMSDCTRXS, description 117
 ARGUMENTS. *See* ARGUMENTS table
 MAP. *See* MAP tables
 ROUTINES. *See* ROUTINES table

technical support xxv

termination
 cleanup by external routine 133
 IMS TM exit routine replacement 119
 of VTAM session 57

TIBCO Enterprise Message Service *See* EMS
 interface 168

TIBCO Object Service Broker
 architecture, introduction to 2
 client model 12
 client styles 12
 clients. *See* client
 processing data within 8
 session types 2
 sessions
 defined 3
 See also session

TIBCO Object Service Broker access statements
 and COBOL programs 340
 checking for runtime errors 348
 coding action statements in COBOL 345
 coding considerations for COBOL 346
 coding operators and expressions 348
 modifying table.field names 347
 naming differences between TIBCO Object Service

- Broker and COBOL 346
- supported statements 348
- writing COBOL program 342
- TIBCO Object Service Broker batch session. *See* batch session
- TIBCO Object Service Broker CICS session. *See* CICS session
- TIBCO Object Service Broker ID
 - default, using as external security user ID 16
- TIBCO Object Service Broker IMS TM session. *See* IMS TM session
- TIBCO Object Service Broker names, renaming to COBOL names 346, 358
- TIBCO Object Service Broker TSO session. *See* TSO session
- TIBCO Object Service Broker user ID
 - using external transaction as 16
- TIBCO Service Gateway for CICS 60
- TIBCO Service Gateway for IMS TM 98
- TIBCO Service Gateway for WMQ, using 196
- tools 49
 - @SESSION
 - and CICS clients 88
 - and the Call Level Interface 251
 - \$GETENVCOMMAREA. *See* \$GETENVCOMMAREA tool
 - \$GETOPT
 - to retrieve IMS TM message information 117
 - to set IMS TM terminal options 105
 - \$SETENVCOMMAREA
 - and the Call Level Interface 205
 - example for CICS client 89
 - example for IMS TM clients 112
 - passing data out, Call Level Interface 249
 - \$SETSESSIONEND
 - and batch clients 39
 - and CICS clients 90
 - and TSO clients 49
 - ENDMSG
 - and CICS COMMAREA 85
 - location of the end message 87
 - GENBIN, and IMS TM 113
 - HLIPREPROCESSOR
 - examples to run 370
 - usage to preprocess TIBCO Object Service Broker
 - or SQL access statements 368
 - MESSAGE_LOG, as used by Call Level Interface 222
 - RETURN_CODE, as used by external routines 134
- trancode
 - supplied for IMS TM session 103
 - used to start IMS TM session 103
- transaction
 - characteristics of 245
 - ending transaction with Call Level Interface 204, 247
 - ensuring consistency in IMC/DC clients 117
 - level of an external routine 133
 - modifying transaction with Call Level Interface 204, 245
 - starting or ending a transaction with Call Level Interface 243
 - starting or ending stream with Call Level Interface 204, 243
 - starting transaction with Call Level Interface 204
- transNestLevel SDK (Java) method 325
- TSO applications, running 42
- TSO client 42–50
 - See also* Call Level Interface; client programs; client styles; client types; Execution Environment parameters; external routines; session parameters
 - address space usage 42
 - and external routines 50
 - CLIST syntax for a session 47
 - invoking *See* starting
 - operation of TIBCO Object Service Broker client 43
 - operation of user written client 45
 - passing data to 49
 - returning data 49
 - sample CLIST 47
 - single-session type 12
 - starting 42
- TSO session
 - authorized and unauthorized libraries 48
 - block size of external routine load library,

- determining 48
- concatenating load library 48
- library concatenation 48
- load library, identifying 48
- search order for external routines 48
- specifying load library 48
- supplied default 46
- types of clients 12

U

- underscore (_), as join character 358
- UNTIL... DISPLAY statement, use in a display
 - client 16
- UPDATE statement, usage with SQL access
 - statements 364
- user builtin routines 160–165
 - samples 163
 - steps required to use 160
- user client
 - types of 13
 - usage of the Call Level Interface 202
- USER CLIST 48
- user ID
 - bypassing with IMS TM client 104
 - determining with a CICS client 66
 - guidelines for setting up 18
 - minimizing number required 16
 - overriding 16
 - setting up profile 18
 - using default 16
- USERID parameter
 - overriding user ID 16
- userId SDK (Java) method 326

V

- variables, for SQL access statements
 - S6B-RETURN-EXCEPTION 361
 - S6B-RETURN-MESSAGE 361
 - Sqlcode 361
 - Sqlstate 361
 - usage of 361
- variables, for TIBCO Object Service Broker access
 - statements
 - HLL-RETURN-CODE 348
 - S6B-RETURN-EXCEPTION 348
 - S6B-RETURN-MESSAGE 348
- VSAM synchronization 69
- VTAM LU2 (3270) terminal
 - logging in to session from 52
 - starting session from 53
 - syntax to establish session with 55
- VTAM LU2 client
 - and Native Execution Environment 52

W

- WebSphere MQ, accessing 196
- WHENEVER statement, usage with SQL access
 - statements 364
- WMQ, gateway for 196
- Working Storage Section, coding TIBCO Object Service Broker access statements in 345
- writeInt SDK (Java) method 334
- writeShort SDK (Java) method 335