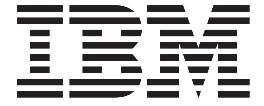
IBM z/VSE

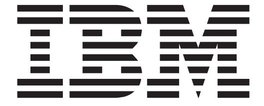


e-business Connectors, User's Guide

Version 5 Release 1

SC34-2629-02

IBM z/VSE



e-business Connectors, User's Guide

Version 5 Release 1

SC34-2629-02

Note: Before using this information and the product it supports, be sure to read the general information under "Notices" on page xv.

This edition applies to Version 5 Release 1 of IBM z/Virtual Storage Extended (z/VSE), Program Number 5609-ZV5, and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SC34–2629–01.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the addresses given below.

A form for readers' comments is provided at the back of this publication. If the form has been removed, address your comments to:

```
IBM Deutschland Research & Development GmbH
Department 3248
Schoenaicher Strasse 220
D-71032 Boeblingen
Federal Republic of Germany
```

You may also send your comments by FAX or via the Internet:

```
Internet: s390id@de.ibm.com
FAX (Germany): 07031-16-3456
FAX (other countries): (+49)+7031-16-3456
```

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright IBM Corporation 2000, 2013.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures ix

Tables xiii

Notices xv
Trademarks xv

Accessibility xvii
Using Assistive Technologies xvii
Documentation Format xvii

About This Publication xix
Who Should Use This Publication. xix
How to Use This Publication xix
Where to Find More Information xix

Summary of Changes xxi

Part 1. INTRODUCTION 1

Chapter 1. Introduction to e-business with z/VSE. 3
Connection Possibilities Using z/VSE e-business Connectors 4
Overview of CICS Connectivity 5
Overview of WebSphere MQ Connectivity 6
Overview of the IBM WebSphere Application Server 6

Chapter 2. Overview of 2- and 3-Tier Environments 7
Overview of 2-Tier Environments 7
Overview of 3-Tier Environments 8
Where to Find Details of the Connectors 9

Part 2. INSTALLATION & CUSTOMIZATION 11

Chapter 3. Choosing the Connectivity You Require 13
Connectivity Possibilities in 2-Tier Environments . . 13
Connectivity Possibilities in 3-Tier Environments . . 14

Chapter 4. Installing the Common Prerequisite Programs 19
Configuring and Activating TCP/IP for VSE/ESA 19
Configuring and Activating the VSE HTTP Server 19
Installing and Configuring Java. 19
Downloading the Java Base Code 20
Deciding Which Java Package to Install 20
Installing the IBM HTTP Server 20
Installing the WebSphere Application Server . . . 21
Installing the WebSphere Application Server on Linux on System z 21
Installing the WebSphere Application Server on z/OS 21
Installing the WebSphere Application Server on Other Platforms 21

Chapter 5. Installing and Operating the Java-Based Connector 23
Overview of the Java-Based Connector 23
Overview of the VSE Connector Client 23
Overview of the VSE Connector Server 24
Installing the VSE Connector Client 24
Obtaining a Copy of the VSE Connector Client 25
Performing the VSE Connector Client Installation 26
Using the Online Documentation Options . . . 26
Configuring for WebSphere Support 27
Uninstalling the VSE Connector Client 27
Configuring the VSE Connector Server 27
Job SKVCSSTJ – Startup Job 28
Job SKVCSCAT – Catalog Members 28
VSE Library Member SKVCSCFG – General Settings. 29
VSE Library Member SKVCSLIB – Specify Libraries to Be Accessed 31
VSE Library Member SKVCSPLG – Specify Plugins to Be Loaded 31
VSE Library Member SKVCSUSR – Specify Logon Access. 32
VSE Library Member SKVCSSSL – Configure for SSL 32
Configuring the Date-Format for the VSE Connector Server 33
Starting the VSE Connector Server. 34
Testing the Communication Between VSE Connector Client and Connector Server. 34
Obtaining a List of VSE Connector Server Commands 35
Entering a Command for the VSE Connector Server 35
Maintaining Security Using the VSE Connector Server 36

Chapter 6. Configuring DL/I for Access Via VSE Java Beans 37
Host Installation Activities That Must Be Already Completed 37
Step 1: Skeleton SKDLISMP – Define Sample Database 37
Step 2: Customize CICS TS 38

Chapter 7. Installing the VSE Script Connector 39
Overview of the VSE Script Connector 39

© Copyright IBM Corp. 2000, 2013

Step 1: Download the Install-File and Perform the Installation 40
Step 1.1: Obtain a Copy of the VSE Script Server 40
Step 1.2: Perform the Installation of the VSE Script Server 41
Step 2: Configure the VSEScriptServer Properties File 42
Step 3: Configure the Connections Properties File. . 43

Chapter 8. Installing the VSAM Redirector Connector 45
Overview of the VSAM Redirector Connector . . . 45
VSAM Integration Considerations 49
Configuring the VSAM Redirector Client / VSAM Capture Exit 49
Step 1: Enable the VSAM Redirector Client / VSAM Capture Exit on z/VSE 50
Step 2: Decide Upon Your Redirection Mode . . 51
Step 3 (Optional): Transfer Your VSAM Data . . 58
Step 4: Create the Configuration Phase 58
Installing the VSAM Redirector Server 63
Step 1: Download the Install-File and Perform the Installation 63
Step 2: Configure the Properties File 65
Step 3: Implement a VSAM Redirector Handler 65
Using the IBM-Provided VSAM Redirector Handlers 67
Using the IBM-Provided VSAM Redirector Loaders 69
Description of the RedirLoader Redirector Loader 69
Description of the MQLoader Redirector Loader 69
Description of the DeltaLoader Redirector Loader 70
IBM-Supplied Example of DB2-Related Handler . . 70

Chapter 9. Installing the Database Call Level Interface 73
Overview of the Database Call Level Interface. . . 73
Overview of Connection Pooling 74
Prerequisites for Using the Database Call Level Interface 74
Installing the DBCLI Server 75
Obtaining a Copy of the DBCLI Server 75
Performing the DBCLI Server Installation . . . 76
Uninstalling the DBCLI Server 76
Setting Up and Configuring the DBCLI Server. . . 76
DatabaseCliServer.cfg 77
JdbcAliases.cfg 78
JdbcDriver.cfg 78
Configuring and Starting/Stopping the Connection Pool Manager 78
Defining CICS Programs/Transactions and OME for Connection Pooling 79
Starting the Connection Pool Manager 79
Stopping the Connection Pool Manager 80
Querying the Connection Pool 81
DBCLI Server Commands 81

Chapter 10. Customizing the DB2-Based Connector 83
Overview of the DB2-Based Connector 83
Host Installation Activities That Must Be Already Completed 84
Step 1: Customize CICS TS 84
Step 2: Customize TCP/IP 85
Step 3: Customize DB2 and Define Sample Database 85
Step 3.1: Define User Catalog 85
Step 3.2: Catalog New ARISIVAR.Z 86
Step 3.3: Job Manager for Preparation / Installation Steps 87
Step 3.4: Activate DRDA Server Support. . . . 87
Step 3.5: Startup Job for Stored Procedure Server 87
Step 3.6: Prepare DB2 Sample Database 88
Step 3.7: Install DB2 Sample Database 89
Step 4: Set Up for DRDA Support 93
Step 5: Set Up Stored Procedure Server and Define to DB2 93
Step 5.1: Set Up the Stored Procedure Server . . 93
Step 5.2: Define Stored Procedure Server to DB2 94
Step 6: Set Up for Stored Procedures 94
Step 7: Customize the DB2-Based Connector for VSAM Data Access 95
Step 8: Customize the DB2-Based Connector for DL/I Data Access 95
Step 9: Start DB2, and Start Stored Procedure Server 96
Step 10: Install DB2 Connect and Establish Client-Host Connection 96

Chapter 11. Configuring the VSAM-Via-CICS Service 99
Configuring the IBM-Supplied CICS System . . . 99
Configuring a Further CICS System for VSAM-Via-CICS 100
How the VSAM-Via-CICS Service Works 101
CICS Transactions for Use with VSAM-Via-CICS 101

Chapter 12. Mapping VSE/VSAM Data to a Relational Structure 103
Introduction to Mapping VSE/VSAM Data . . . 103
How VSAM Maps Are Structured 104
How Maps Are Stored on the z/VSE host 104
Defining a Map Using RECMAP 105
Defining a Map Using the Sample Applet 105
Defining a Map Using a Java Application 106
Defining a Map Using the VSAM MapTool . . . 112

Part 3. SYSTEM MONITORING . . . 113

Chapter 13. Collecting Data via the VSE Monitoring Agent. 115
Overview of How VSE Data is Collected 115
Configuring the VSE Monitoring Agent. 117
Maximizing the Security of SNMP Communication 118
Configuring the IBM-Supplied Monitoring Plug-ins 119
Commands Available With the VSE Monitoring Agent 120
Examples of How to Collect Data 121
Using the SNMP Trap Client in Batch Jobs . . . 121
Using the Trap Client API in LE/C Programs. . . 123
Functions Provided by the Trap LE/C interface . . 124
Using the Trap Client API in COBOL and PL/I Programs. 124

Using the Trap Client API in CICS Programs 125
Copybook Used for the Trap COBOL and PL/I Batch interface 126
Return Codes Generated by the Trap Client API 127
Creating Your Own Plug-Ins for the VSE Monitoring Agent 127

Chapter 14. Using GDPS Support for High Availability 129
Overview of How the GDPS Client Is Used 129
Configuring the GDPS Client 130
Starting the GDPS Client 131
Communication Between GDPS Client and GDPS K-System 131
Commands Available With the GDPS Client 132

Part 4. PROGRAMMING 135

Chapter 15. Using VSE Java Beans to Implement Java Programs 141
Where VSE Java Beans Are Installed and Used . . 142
How JavaBeans and EJBs Compare to VSE Java Beans 142
Contents of the VSE Java Beans Class Library . . 143
Example of a Javadoc for a VSE Java Bean 145
Using the Callback Mechanism of VSE Java Beans 146
Example of Using VSE Java Beans to Connect to the Host 148
  Step 1: Create a VSEConnectionSpec. 149
  Step 2: Create a VSESystem 149
Example of Using VSE Java Beans to Connect to the Host via SSL 149
  Step 1: Prompt for IP Address, User ID, Password. 150
  Step 2: Create a Connection Specification for the VSE System 150
  Specifying the SSL Properties: Alternative 1 . . 150
  Specifying the SSL Properties: Alternative 2 . . 151
  Specifying the SSL Properties: Alternative 3 . . 151
  Main Method of the Class Used in This Example 152
  Implementation of the ConfirmCertificate Method 152
Example of Using VSE Java Beans to Submit Jobs to the Host 152
  Step 1: Prompt for IP Address, User ID, Password. 153
  Step 2: Create a Connection Specification for the VSE System 153
  Step 3: Submit a Job File. 153
  Step 4: Create the Job File and Send It to the Host 153
Example of Using VSE Java Beans to Access the Operator Console 155
  Step 1: Prompt for IP Address, User ID, Password. 155
  Step 2: Create a Connection Specification for the VSE System 155
  Step 3: Create a Console Instance and Send a Command 156
  Step 4: Obtain Messages One Line at-a-Time . . 156
Example of Using VSE Java Beans to Access VSAM Data 156
  Step 1: Define the Local Variables 157
  Step 2: Prompt for IP Address of VSE System, User ID, Password 157
  Step 3: Create Connection Specification for the VSE System 157
  Step 4: Create a VSEResourceListener 157
  Step 5: Get VSAM Records from the z/VSE Host 158
  Step 6: Display VSAM Records 158
  Step 7: Insert a VSAM Record in the VSAM Cluster 158
  Step 8: Prompt the User to Enter Column Values 159
Example of Using VSE Java Beans to Access DL/I Data 159
  Step 1: Create a VSE System Instance and Get Access to DL/I 160
  Step 2: Schedule the PSB 160
  Step 3: Get a PCB 160
  Step 4: List DL/I Segments. 161
  Step 5: Insert or Update a DL/I Segment . . . 161
  Step 6: Delete a DL/I Segment 163
  Step 7: Terminate the PSB 163
Example of Using VSE Java Beans to Access VSE/POWER Data 163
  Step 1: Prompt for IP address, User ID, and Password. 164
  Step 2: Create a Connection Specification for the VSE System 164
  Step 3: Create a VSEResourceListener 164
  Step 4: Scan Compile Outputs for Errors . . . 165
Example of Using VSE Java Beans to Access Librarian Data 165
  Step 1: Prompt for IP address, User ID, and Password. 165
  Step 2: Create a Connection Specification for the VSE System 166
  Step 3: Create a VSEResourceListener 166
  Step 4: Obtain a List of Libraries From the VSEResourceListener 166
  Step 5: Obtain and Count a List of Sub-Libraries 167
  Step 6: Obtain the Instance of the PRD2.CONFIG Sub-Library 167
  Step 7: Obtain a List of Members in the PRD2.CONFIG Sub-Library 167
  Step 8: Obtain Properties of the First Member 167
  Step 9: Download the Member to a Local Disk 168
Example of Using VSE Java Beans to Access VSE/ICCF Data 168
  Step 1: Prompt for IP address, User ID, and Password. 168
  Step 2: Create a Connection Specification for the VSE System 168
  Step 3: Create a VSEResourceListener 169
  Step 4: Obtain a List of ICCF Libraries From the VSEResourceListener 169
  Step 5: Download a Specific VSE/ICCF Member 170
  Step 6: Download a Specific VSE/ICCF Member (Very Fast Method) 170

Using the VSE Navigator Application 170
Prerequisite for Using the VSE Navigator . . . 172
Migrating From Earlier Versions 172
Installing the VSE Navigator 173
Starting the VSE Navigator Client 173
Adding Your Own VSE Navigator Plug-Ins . . 174
Using the VSE Health Checker Application . . . 175
Prerequisite for Using the VSE Health Checker 176
Migrating From Earlier Versions 176
Installing the VSE Health Checker 177
Starting the VSE Health Checker Client. . . . 177

Chapter 16. Using JDBC to Access VSAM Data 179
SQL Statements That Are Supported by JDBC . . 179
Relational and VSE Java Beans Terminology . . . 182
Specifying Table Names 182
Example of Using JDBC to Access VSAM Data . . 182
Step 1. Define the Local Variables 183
Step 2: Prompt for IP address, User ID, and Password. 183
Step 3. Establish a Connection to the z/VSE Host 183
Step 4. Display a List of Rows in the Database 184
Step 5. Process Result-Set Returned From JDBC 184
Step 6. Add a New Record 185

Chapter 17. Using Java Applets to Access Data 187
How Applets Are Used in 2-Tier Environments . . 187
How Applets Are Used in 3-Tier Environments . . 189
How the VSEAppletServer Is Used 191
Disadvantages and Restrictions Of Using Applets 191
Running the Sample Data-Mapping Applet . . . 192
Description of the Data-Mapping Applet . . . 192
Activities Required on the z/VSE Host 194
Deploying the Data-Mapping Applet 194
Calling the Data-Mapping Applet 195
Setting Up the Data-Mapping Applet Class . . 195
Initializing the Data-Mapping Applet 196
Re-Displaying or Leaving an HTML Page . . . 196
Using the Data-Mapping Applet to Add a Map to a VSAM Cluster 196
Using the Data-Mapping Applet to Modify a Map 197
Using the Data-Mapping Applet to Modify a Map's Data Fields 198
Running the Data-Mapping Applet Locally Using the AppletViewer 200
Running the Sample VSAM Applet 201
Description of the VSAM Applet 201
Getting Started With the Sample VSAM Applet 203
Calling the VSAM Applet 206
Description of DB2ConnectorJDBCApplet.java (the Client-Side Program) 208
Description of VSAMSEL 210
Running the Sample DL/I Applet 214
Description of the DL/I Applet 214
Getting Started With the Sample DL/I Applet 216
Calling the DL/I Applet. 218
Description of DB2DLIConnectorJDBCApplet.java (the Client-Side Program) 220
Description of DLIREAD 222

Chapter 18. Using Java Servlets to Access Data 227
How Servlets Are Used in 3-Tier Environments . . 227
Compiling and Calling Servlets 229
How the WebSphere Application Server Stores Session Information 230
Example of How to Implement a Servlet 230
General Description of the Sample Servlet . . . 231
Creating the VSAM Clusters for the Sample . . 231
HTML Constructs Used With the Sample . . . 232

Chapter 19. Using Java Server Pages to Access Data. 245
How JSPs Are Used in 3-Tier Environments . . . 245
Example of a Simple Java Server Page 247

Chapter 20. Using EJBs to Represent Data 249
Overview of the EJB Architecture. 249
Overview of How EJB Containers are Used . . 250
How EJBs Compare to JavaBeans / Java Servlets 251
Implementing Your Client Applications. . . . 252
How an EJB Client Accesses EJBs. 253
Example of Using EJBs to Access VSAM Data . . 254
Example of Implementing VSAM-Based EJBs. . . 256
Step 1: Define the Sample's VSAM Cluster . . 257
Step 2: Create the Record Layout for Employees 257
Step 3: Specify the EJB's Home Interface . . . 258
Step 4: Specify the EJB's Remote Interface . . . 258
Step 5: Implement the RecordPK Class 259
Step 6: Implement the EJB Code 259
Step 7: Compile the Java Source Files 264
Step 8: Deploy the EJBs 264
Step 9: Access the EJBs from an EJB Client . . 265

Chapter 21. Extending the Java-Based Connector 269
Implementing a Server Plugin 269
Implementing a PluginMainEntryPoint Function 274
Implementing a SetupPlugin Function 275
Implementing a CleanupPlugin Function . . . 276
Implementing a GetHandledCommands Function 277
Implementing a SetupHandler Function . . . 278
Implementing an ExecuteHandler Function . . 279
Implementing a CleanupHandler Function . . 281
Creating Your Own Plugin Callback Functions 282
Action Codes Supported by the VSE Connector Server 282
Utility Functions Supported by the VSE Connector Server 284
Using the IBM-Supplied Server Plugin Example 285
Registering and Compiling Your Server Plugin 285
Implementing a Client Plugin 286

Using the VSEPlugin class 286
General Considerations When Designing Your Plugin 288
Specifying the Protocol Between VSE Connector Server and Plugin 288
Choosing the Access Method to the Data / Application 289
Considerations for ASCII / EBCDIC and Big / Little Endian 289
Deciding Which Requests / Functions Should Be Supported 290
Transferring Data Over the Network 290
Structuring the Client Plugin's View 290

Chapter 22. Using the Database Call Level Interface to Access Data 291

DBCLI Programming Concepts 291
Initializing and Terminating the API Environment 291
Connecting and Disconnecting to/from the DBCLI Server and Vendor Database 292
Logical Units of Work (Transactions) 292
Executing SQL Statements 293
Cursors 294
Database Meta Data 295
Connection Pooling 295
Programming Restrictions and Requirements . . . 296
Using the DBCLI in COBOL 296
Using The DBCLI in PL/I 297
Using The DBCLI in C 297
Using the DBCLI in Assembler 298
Using the DBCLI in REXX 299
Syntax and Parameters of a DBCLI Function Call 299
DBCLI Functions (Reference Information) 300
Roadmap of Where the DBCLI Functions Are Used 300
BINDCOLUMN 301
BINDPARAMETER 304
CLOSECURSOR 307
CLOSESTATEMENT 308
COMMIT 308
CONNECT 309
CONNECTSSL 311
DBATTRIBUTES 313
DBBESTROWIDENT 315
DBCATALOGS 317
DBCOLUMNPRIV 318
DBCOLUMNS 320
DBCROSSREFERENCE 322
DBEXPORTEDKEYS 325
DBIMPORTEDKEYS 327
DBINDEXINFO 329
DBPRIMARYKEYS 332
DBPROCEDURECOLS 333
DBPROCEDURES 335
DBSCHEMAS 337
DBSUPERTABLES 338
DBSUPERTYPES 339
DBTABLEPRIV 341
DBTABLES 343
DBTABLETYPES 345
DBTYPEINFO 346
DBUDTS 347
DBVERSIONCOLS 349
DISCONNECT 350
EXECUTE 351
FETCH 352
GETCOLUMNINFO 353
GETCONNATTR 355
GETENVATTR 369
GETLASTERROR 371
GETMORERESULTS 372
GETNUMCOLUMNS 373
GETNUMPARAMETERS 374
GETPARAMETERINFO 374
GETROWNUMBER 376
GETSTMTATTR 376
GETUPDATECOUNT 378
INITENV 379
INITSSL 380
PREPARECALL 381
PREPARESTATEMENT 382
RELEASESAVEPOINT 384
ROLLBACK 385
SETCONNATTR 385
SETENVATTR 386
SETPOS 387
SETSAVEPOINT 388
SETSTMTATTR 389
TERMENV 389
Performance Considerations When Using the DBCLI 390
Using SSL 390
Pre-fetching 390
Binding Columns 391
Investigating the Cause of Errors When Using the DBCLI 391
Return Codes Used by the DBCLI 392

Chapter 23. Using the DB2-Based Connector to Access Data 393

How You Use DB2 Stored Procedures 393
Grouping Stored Procedure Servers 394
Programming Requirements When Using DB2 Stored Procedures 394
Using DB2 Stored Procedures to Access VSAM Data 395
Overview: Accessing VSAM Data via DB2 Stored Procedures 395
Using the Call Level Interface: Activities on the Requestor 397
Using Call Level Interface: Activities on the z/VSE host 398
Example of the Syntax of a CLI Function – VSAMSQLCloseTable 399
Program Flow When Using the VSAMSQL Call Level Interface 399
SQL Statements Supported by VSAMSQL Call Level Interface 400
Using DB2 Stored Procedures to Access DL/I Data 402
Overview of the AIBTDLI Interface 403
Creating Programs That Use AIBTDLI 405

Invoking the AIBTDLI Interface 406
Compiling and Link-Editing Your Programs . . 408
Return and Status Codes 408
Scheduling with Single and Multiple MPS Systems 409
Task Termination and Abend Handling. . . . 410
Messages and Return Codes 411

Chapter 24. Using SOAP for Inter-Program Communication 413

Overview of z/VSE Support for Web Services and SOAP 414
Overview of the SOAP Syntax. 414
Overview of Web Service (SOAP) Security. . . . 415
Comparison of Transport-Layer Security and Message-Layer Security 415
Using Authentication With Web Service Security 416
How the z/VSE Host Can Act As the SOAP Server 418
Using Web Service Security Features When z/VSE Acts As the SOAP Server 419
How the z/VSE Host Can Act As the SOAP Client 420
Using Web Service Security Features When z/VSE Acts As the SOAP Client 421
How the IBM-Supplied SOAP Control Blocks Are Used 421
How the SOAP_PARAM_HDR Control Block Is Used 422
How the SOAP_PROG_PARAM Control Block Is Used 424
How the SOAP_DEC_PARAM Control Block Is Used 425
Configuring the z/VSE SOAP Engine 426
Mapping Long-Names to Short-Names 428
Description of the IBM-Supplied SOAP Service (getquote.c) 428
Description of the IBM-Supplied SOAP Client (soapclnt.c) 430
Using a Java SOAP Client 432
Running the IBM-Supplied SOAP Sample 433
Step 1: Download and Install the Java SOAP Client Packages on the Client 433
Step 2: Extract and Install the Required Java Programs. 433
Step 3: Compile /Link the Sample C Programs, and Define Them to CICS 434
Step 4: Configure CICS to Use CICS Web Support 435
Step 5: Define the SOAP Server to CICS . . . 435
Step 5: Activate the ASCII to EBCDIC Converter 435
Step 6: Compile the Java Sample 435
Step 7: Run the Java SOAP Client Sample . . . 436
Step 8: Run the C-Program SOAP Client Sample 436
Writing Your Own SOAP Programs 436

Chapter 25. Using the VSE Script Connector for Non-Java Access . . . 437

How the VSE Script Connector Can Be Used . . . 437
Overview of the Protocol Used Between Client and Server 439
Using SSL-Encrypted Connections 440
Writing VSE Scripts Using the VSE Script Language. 440
General Rules That Apply to the VSE Script Language. 440
VSE Script Language Built-In General Functions 441
VSE Script Language: Built-In String Functions 442
VSE Script Language: Built-In Console Functions. 443
VSE Script Language: Built-In POWER Functions. 443
VSE Script Language: Librarian Functions . . . 443
VSE Script Language: Built-In VSAM Functions 444
Sample Files You Can Use for Writing VSE Script Clients 445
Example of Writing a VSE Script Client (and Its VSE Script) 445
Step 1: Setup the VSE Script Server Properties File. 446
Step 2: Setup the Connections Properties File 446
Step 3: Define the Sample VSAM Data 446
Step 4: Modify the Sample VSE Script 447
Step 5: Start the VSE Connector Server on the z/VSE Host 448
Step 6: Start the VSE Script Server Locally. . . 449
Step 7(a): Open the Sample Lotus 1-2-3 Spreadsheet File 449
Step 7(b): Open the Sample MS Office Spreadsheet 452
Step 7(c): Start a Sample VSE Script from the Command Line. 455
Obtaining Data From the Middle-Tier Using VSE Script Clients 455
Using the VSE Script Client That Runs in Batch 457
Using the VSE Script Client That Runs Under CICS 459

Part 5. Appendixes 463

Glossary 465

Index 471

Figures

1. Overview of Connection Possibilities under z/VSE. 5
2. Overview of 2-Tier Environments and the Programs You Can Use 8
3. Overview of 3-Tier Environments and the Programs You Can Use 9
4. Job SKVCSSTJ (for placing Startup Job in Reader Queue) 28
5. Job SKVCSCAT (for Cataloging Members for VSE Connector Server). 29
6. Member SKVCSCFG (for Specifying General Settings for VSE Connector Server). 29
7. Member SKVCSLIB (for Specifying Libraries to Be Accessed by VSE Connector Server) . . . 31
8. Member SKVCSPLG (for Specifying Plugins for VSE Connector Server) 31
9. Member SKVCSUSR (for Specifying Logon Access to VSE Connector Server) 32
10. Member SKVCSSSL (for Configuring the VSE Connector Server for SSL). 33
11. Startup Job STARTVCS (for Starting the VSE Connector Server) 34
12. Displaying the Commands Provided by the VSE Connector Server 35
13. How Synchronous Data Redirection Is Used 47
14. How Asynchronous Data Redirection Is Used 48
15. Flow of Control for VSAM PUT Request 52
16. Flow of Control for VSAM GET Request 53
17. Flow of Control for VSAM PUT Request 54
18. Flow of Control for VSAM GET Request 54
19. Job to Produce a Configuration Phase for the VSAM Redirector Connector 62
20. Hierarchical Structure of VSAM Maps 104
21. Job To Define the Cluster for VSE.VSAM.RECORD.MAPPING.DEFS . . . 105
22. Example of Creating a Local VSAM Map Object 107
23. Example of Creating Data Fields for a Map 108
24. Example of Displaying the Properties of a Map 108
25. Example of Creating a View for a Map 109
26. Example of Adding Data Fields to a View 109
27. Example of Displaying the Properties of a View 110
28. Example of How to Delete a Map. 110
29. Example of a VSAM MapTool Window 112
30. Sending SNMP Traps to the VSE Monitoring Agent 116
31. Job to Start the VSE Monitoring Agent 117
32. Sample Configuration File Used by the VSE Monitoring Agent 117
33. Job to Replace the Configuration File Used by the VSE Monitoring Agent 118
34. Configuration File Belonging to the System Plug-In. 119
35. Job to Replace the Configuration File Used by the System Plug-In. 119
36. Listing of Commands That Can Be Used With the VSE Monitoring Agent 120
37. Obtaining the Status of the VSE Monitoring Agent 120
38. Job to Start the SNMP Trap Client in a Batch Job 122
39. Listing of Parameters That Can Be Used With the SNMP Trap Client. 123
40. Copybook Used as Input for the Trap COBOL and PL/I Batch interface. 126
41. Overview of z/VSE's GDPS Support 129
42. Configuring the GDPS Client Using Job SKGDPSCF 130
43. Sample Job SKSTGDPS to Start the GDPS Client 131
44. Example of a Javadoc Belonging to the VSE Java Beans Class Library. 145
45. Program Flow for Using VSE Java Classes to Obtain a List of VSAM Catalogs 148
46. Connect to Host via VSE Java Beans: Create a VSEConnectionSpec 149
47. Connect to Host via VSE Java Beans: Create a VSESystem 149
48. Connect to Host via VSE Java Beans and SSL: Prompt for IP Address, User ID, Password. . 150
49. Connect to Host via SSL and VSE Java Beans: Create a Connection Specification. 150
50. Connect to Host via VSE Java Beans and SSL: Specify SSL Properties (Alternative 1) . . . 151
51. Connect to Host via VSE Java Beans and SSL: Specify SSL Properties (Alternative 2) . . . 151
52. Connect to Host via VSE Java Beans and SSL: Specify SSL Properties (Alternative 3) . . . 151
53. Connect to Host via VSE Java Beans and SSL: Main Method of the Class 152
54. Connect to Host via VSE Java Beans and SSL: Implementation of ConfirmCertificate Method 152
55. Submit Jobs via VSE Java Beans: Create a Connection Specification. 153
56. Submit Jobs via VSE Java Beans: Submit a Job File 153
57. Submit Jobs via VSE Java Beans: Create the Job File and Send to the Host 154
58. Access Console via VSE Java Beans: Create a Connection Specification. 155
59. Access Console via VSE Java Beans: Create Console Instance, Send a Command 156
60. Access Console via VSE Java Beans: Obtain Messages One Line at-a-Time 156
61. VSAM Data via VSE Java Beans: Define Local Variables 157
62. VSAM Data via VSE Java Beans: Create Connection Specification. 157

© Copyright IBM Corp. 2000, 2013

63. VSAM Data via VSE Java Beans: Create a VSEResourceListener 157
64. VSAM Data via VSE Java Beans: Get VSAM Records from Host 158
65. VSAM Data via VSE Java Beans: Display VSAM Records 158
66. VSAM Data via VSE Java Beans: Insert a VSAM Record 159
67. VSAM Data via VSE Java Beans: Prompt the User for Values 159
68. DL/I Data via VSE Java Beans: Get Access to DL/I 160
69. DL/I Data via VSE Java Beans: Schedule the PSB 160
70. DL/I Data via VSE Java Beans: Get a PCB 160
71. DL/I Data via VSE Java Beans: List DL/I Segments 161
72. DL/I Data via VSE Java Beans: Insert/Update a DL/I Segment 162
73. DL/I Data via VSE Java Beans: Delete a DL/I Segment 163
74. DL/I Data via VSE Java Beans: Terminate the PSB 163
75. VSE/POWER Data via VSE Java Beans: Create a Connection Specification 164
76. VSE/POWER Data via VSE Java Beans: Create a VSEResourceListener 164
77. VSE/POWER Data via VSE Java Beans: Scan for Compile Errors 165
78. Librarian Data via VSE Java Beans: Create a Connection Specification 166
79. Librarian Data via VSE Java Beans: Create a VSEResourceListener 166
80. Librarian Data via VSE Java Beans: Obtain a List of Libraries 166
81. Librarian Data via VSE Java Beans: Obtain/Count a List of Sub-Libraries . . . 167
82. Librarian Data via VSE Java Beans: Get the Instance of the Sub-library 167
83. Librarian Data via VSE Java Beans: Obtain a List of Members in PRD2.CONFIG 167
84. Librarian Data via VSE Java Beans: Obtain a List of Members in PRD2.CONFIG 167
85. Librarian Data via VSE Java Beans: Download the Member to Disk 168
86. VSE/ICCF Data via VSE Java Beans: Create a Connection Specification 169
87. VSE/ICCF Data via VSE Java Beans: Create a VSEResourceListener 169
88. VSE/ICCF Data via VSE Java Beans: Obtain a List of ICCF Libraries 169
89. VSE/ICCF Data via VSE Java Beans: Download a Specific Member 170
90. VSE/ICCF Data via VSE Java Beans: Download a Specific Member (Very Fast) . . 170
91. Displaying System Activity Using the VSE Navigator 171
92. Displaying VSAM Files Using the VSE Navigator 172
93. Configure Hosts for the VSE Navigator 174
94. Using the VSE Navigator to Access CICS Data 175
95. Graphical User Interface, as Provided by the VSE Health Checker 176
96. VSAM Data via JDBC: Define Local Variables 183
97. VSAM Data via JDBC: Prompt for IP Address, User ID, Password 183
98. VSAM Data via JDBC: Establish a Host Connection 184
99. VSAM Data via JDBC: Display the Database Rows 184
100. VSAM Data via JDBC: Process Result-Set 184
101. VSAM Data via JDBC: Add a New Record 185
102. How Applets Are Used in the z/VSE 2-Tier Environment 188
103. How Applets Are Used in the z/VSE 3-Tier Environment 190
104. How the VSEApplet Server Is Used in the 3-Tier Environment 191
105. Window for VSAM Data-Mapping Applet 193
106. Data-Mapping Applet Code for Setting Up the Java Class 195
107. Sample Code for Initializing the Data-Mapping Applet. 196
108. Data-Mapping Applet Code for Adding a Map to a VSAM Cluster 197
109. Sample Applet Code for Modifying a Map 197
110. Window for Changing a Map's Properties 198
111. Sample Applet Code for Modifying a Map's Data Fields 199
112. Window for Changing a Map's Data Fields 200
113. Using the Sample VSAM Applet to Access VSAM Data 202
114. Window Displayed by the Sample VSAM Applet 207
115. Using the Sample DL/I Applet to Access DL/I Data 215
116. Window Displayed by the Sample DL/I Applet 219
117. How Servlets Are Used in the z/VSE 3-Tier Environment 228
118. How Session Information Is Re-Used by the WebSphere Application Server 230
119. VSAM Structure of FLIGHT.ORDERING.FLIGHTS 231
120. VSAM Structure of FLIGHT.ORDERING.ORDERS 232
121. Example of a Servlet Using Forms to Obtain a User's Input 233
122. Example of Using Forms to Display Window Controls 233
123. Sample Servlet Code for Displaying a List of Flights 234
124. Sample Servlet Code for Getting Flight Instances from the Host 235
125. Flight Order Selection Window, As Generated by the Sample Servlet. 236
126. Sample Servlet Code for Displaying Properties of a Flight 237
127. Flight Order Entry Window, As Generated by the Sample Servlet 238

128. Sample Servlet Code for Creating a New Flight 241
129. Sample Servlet Code for Creating a New Order 242
130. Flight Order Confirmation Window, Generated by the Sample Servlet 243
131. How JSPs Are Used in the z/VSE 3-Tier Environment. 246
132. Example of a Java Server Page /JSP) 247
133. How Containers Are Used To Manage EJBs 251
134. Overview of the Entities Involved in an EJB Method Call 252
135. How an EJB Client Communicates with an EJB 253
136. How EJBs Are Used Together with an Applet in the 3-Tier Environment 255
137. How the EJB Client Accesses EJBs in the Provided Example 266
138. Example of EJB Client Code 267
139. Overview of How a Server Plugin's Functions Are Called 270
140. How a Plugin's Functions Are Called During Startup. 271
141. How a Plugin's Functions Are Called When a Request Is Received 272
142. Overview of How a Plugin's Functions Are Called During Server Shutdown 273
143. Sample Code for Implementing PluginMainEntryPoint Function 274
144. Sample Code for Implementing the SetupPlugin Function 276
145. Sample Code for Implementing the CleanupPlugin Function 277
146. Sample Code for Implementing the GetHandledCommands Function 278
147. Sample Code for Implementing the SetupHandler Function 279
148. Sample Code for Implementing the ExecuteHandler Function 280
149. Sample Code for Distinguishing Between Multiple Requests Within a Plugin 281
150. Example of Calling a Stub Code Written in a Language Other Than C 282
151. How You Use DB2 Stored Procedures To Access VSAM Data 396
152. Typical Program Flow When Performing a VSAMSQL CLI Update 400
153. How You Use DB2 Stored Procedures To Access DL/I Data 402
154. DL/I Partition Layout for Batch, MPS Batch, CICS/DLI Online, and AIBTDLI Interface . . 404
155. Connections Between Web Service Requester and Web Service Provider 415
156. How SOAP Is Used When the z/VSE Host Acts As SOAP Server 418
157. How SOAP Is Used When the z/VSE Host Acts As SOAP Client 420
158. Contents of the SOAP Parameter 422
159. Possible Values for `type` of Value Field 422
160. Fields Contained in SOAP_PROG_PARAM Control Block 424
161. Fields Contained in SOAP_DEC_PARAM Control Block 425
162. Proxy Types That Can Be Used With SOAP_DEC_PARAM Control Block 426
163. Mapping COMMAREA to SOAP_PROG_PARAM Control Block . . . 429
164. Checking Which SOAP Method Has Been Requested. 429
165. Get Input Parameters from CICS Queue 429
166. Put Parameter Into the CICS Output Queue 430
167. Preparing the SOAP Client's Call Parameter 431
168. SOAP Client Prepares the SOAP_DEC_PARAM Structure. 431
169. SOAP Client Inserts Values into the SOAP Server's Input Queue 431
170. SOAP Client Calls SOAP Converter (IESSOAPE) to Handle Requests 431
171. SOAP Client Obtains Results of the SOAP Call 431
172. SOAP Client Deletes CICS Queues 432
173. SOAP Client Calls the getQuote Service 432
174. Using CEDA to Define Sample SOAP Service to CICS 434
175. Using CEDA to Define SOAP Server to CICS 435
176. Obtaining Data From the Host or Middle-Tier Using the VSE Script Connector 438
177. VSE Script Provided With the VSE Script Connector Example 448
178. Sample Lotus 1-2-3 Spreadsheet for VSE Script Connector Example 449
179. Transferring Data from VSAM Cluster to Lotus 1-2-3 Spreadsheet 450
180. Sample Script As Defined in Lotus 1-2-3 450
181. Visual Basic Script Used With Lotus 1-2-3 Spreadsheet Example 451
182. Sample Spreadsheet for MS Office Spreadsheet Example 452
183. Transferring Data from VSAM Cluster to MS Office Spreadsheet 453
184. Sample Script as Defined in MS Office 453

Tables

1. Connectivity Possibilities in 2-Tier Environments 13
2. Connectivity Possibilities in 3-Tier Environments 15
3. Currently-Supplied VSAM Redirector Handlers 67
4. Contents of the VSE Java Beans Class Library 143
5. SQL Statements Supported by JDBC 179
6. Relational Terms and Their VSE Equivalents 182
7. Properties of Session Beans and Entity Beans 249
8. Roadmap of Where DBCLI Functions Are Used 300
9. CLI Functions You Can Use for Accessing Mapped VSAM Data 398
10. Files Supplied for Writing VSE Script Clients 445

© Copyright IBM Corp. 2000, 2013

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of the intellectual property rights of IBM may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785, U.S.A.

Any pointers in this publication to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement. IBM accepts no responsibility for the content or use of non-IBM websites specifically mentioned in this publication or accessed through an IBM website that is mentioned in this publication.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

```
IBM Deutschland GmbH
Dept. M358
IBM-Allee 1
71139 Ehningen
Germany
```

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

IPv6/VSE is a registered trademark of Barnard Software, Inc.

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/VSE enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

Using Assistive Technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/VSE. Consult the assistive technology documentation for specific information when using such products to access z/VSE interfaces.

Documentation Format

The publications for this product are in Adobe Portable Document Format (PDF) and should be compliant with accessibility standards. If you experience difficulties when you use the PDF files and want to request a web-based format for a publication, you can either write an email to s390id@de.ibm.com, or use the Reader Comment Form in the back of this publication or direct your mail to the following address:

```
IBM Deutschland Research & Development GmbH
Department 3282
Schoenaicher Strasse 220
D-71032 Boeblingen
Federal Republic of Germany
```

In the request, be sure to include the publication number and title.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright IBM Corp. 2000, 2013

About This Publication

This publication describes how you can use the z/VSE e-business Connectors to develop e-business applications that access programs and data based on the z/VSE host.

Who Should Use This Publication

This publication is intended for systems programmers who install additional z/VSE programs, and application programmers who are familiar with the Java™ object-oriented programming language.

How to Use This Publication

This publication is divided into four parts, as follows.

- **Part 1** provides you with an introduction to the z/VSE e-business Connectors, and the basic possible configurations of e-business solutions.
- **Part 2** describes the installation and customization activities you might carry out, in order to establish e-business connectivity within your z/VSE system.
- **Part 3** provides theoretical and practical information describing how to use the Java-based connector and DB2-based connector to develop your e-business applications.

Where to Find More Information

Here is a list of IBM® publications and publications from other vendors, that you might find useful. The IBM redbooks listed here are usually not kept up-to-date, but on the other hand *at the time they are written* they are at the forefront of the technical areas they describe.

- *z/VSE Planning*, SC34-2635
- *z/VSE TCP/IP Support*, SC34-2640
- *CICS Transaction Server for VSE/ESA, Enhancements Guide*, GC34-5763
- *WebSphere MQ for z/VSE System Management Guide*, GC34-6981.
- *WebSphere V5 for Linux on zSeries® Connectivity Handbook (Redbook)*, SG24-7042
- *e-business Solutions for z/VSE (Redbook)*, SG24-5662
- *WebSphere Application Servers: Standard and Advanced Editions (Redbook)*, SG24-5460
- *Security on IBM z/VSE (Redbook)*, SG24-7691

The online documentation provided by the VSE Connector Client includes a list of useful Internet sites, and online publications. See "Using the Online Documentation Options" on page 26 for a description.

In addition, you might refer to the ...

z/VSE Home Page

z/VSE has a home page on the World Wide Web, which offers up-to-date information about VSE-related products and services, new z/VSE functions, and other items of interest to VSE users.

© Copyright IBM Corp. 2000, 2013

You can find the z/VSE home page at

http://www.ibm.com/systems/z/os/zvse/

You can also find VSE User Examples (in zipped format) at

http://www.ibm.com/systems/z/os/zvse/downloads/samples.html

Summary of Changes

This is the enhancement made available via the *June 2013* Service Upgrade of z/VSE 5.1:

- The Database Call Level Interface now supports *connection pooling*. See "Overview of Connection Pooling" on page 74 and "Configuring and Starting/Stopping the Connection Pool Manager" on page 78.

This is the enhancement made available via the *June 2012* Service Upgrade of z/VSE 5.1:

- The *Database Call Level Interface* is a database connector that allows z/VSE applications to access a relational database (IBM DB2®, Oracle, Microsoft SQL Server, MySQL, and so on) on any suitable database server. See Chapter 9, "Installing the Database Call Level Interface," on page 73 and Chapter 22, "Using the Database Call Level Interface to Access Data," on page 291.

These are the new items and changes delivered at *General Availability* of z/VSE 5.1:

- The VSE Script connector now supports these additional functions:
  - These code page tools have been added to "VSE Script Language: Built-In String Functions" on page 442:
    - binary to string
    - string to binary
    - save file binary
    - read file binary.
  - These VSE/POWER functions have been added to "VSE Script Language: Built-In POWER Functions" on page 443:
    - binary Get function
    - binary Put function.
  - The Librarian functions described in "VSE Script Language: Librarian Functions" on page 443 are new.
- The VSE Monitoring Agent has been enhanced to allow your COBOL, PL/I, and C/VSE programs, and CICS® transactions to *dynamically* link the SNMP Trap Client API to enable them to send SNMP "traps". See Chapter 13, "Collecting Data via the VSE Monitoring Agent," on page 115.
- When using the SNMP Trap Client in batch jobs, you can now include *symbolic parameters*. See "Using the SNMP Trap Client in Batch Jobs" on page 121.
- GDPS® support is new. A GDPS Client can collect availability data from your z/VSE systems and send this data to a central GDPS K-System. See Chapter 14, "Using GDPS Support for High Availability," on page 129.
- You can now use *symbolic parameters* when running the VSE Script Client in batch. See "Using the VSE Script Client That Runs in Batch" on page 457.
- The current versions of these products are supported:
  - Windows (for example, Windows Vista) together with the related security concepts.
  - Java, together with the related SSL functionality.
  - TCP/IP.

© Copyright IBM Corp. 2000, 2013

- WebSphere®.

Note: For an overview of *all* the items introduced with z/VSE 5.1, refer to *z/VSE Release Guide*, SC34-2636.

Part 1. INTRODUCTION

Chapter 1. Introduction to e-business with z/VSE 3
Connection Possibilities Using z/VSE e-business Connectors 4
Overview of CICS Connectivity 5
Overview of WebSphere MQ Connectivity 6
Overview of the IBM WebSphere Application Server 6

Chapter 2. Overview of 2- and 3-Tier Environments 7
Overview of 2-Tier Environments 7
Overview of 3-Tier Environments 8
Where to Find Details of the Connectors 9

© Copyright IBM Corp. 2000, 2013

Chapter 1. Introduction to e-business with z/VSE

To help you be successful in meeting your Internet and e-business requirements, IBM has developed an *Application Framework for e-business*. This framework has been developed with the aim of helping you to protect your investment in existing information assets, while enabling you to exploit the emerging e-business opportunities. The Application Framework for e-business consists of three logical layers:

1. A client, which is usually a workstation that has a standard Web browser installed. However, this can also be a wireless telephone or Personal Digital Assistant (PDA).
2. A Web application server, which is the "hub" that processes requests from clients, controls access to business logic and data. If the logic or data are stored on a different system, the Web application server uses connectors to access them. The Web application server also integrates static and dynamic content, and then returns Web pages to clients.
3. *Connectors*, which provide access to external services such as business logic and data. The connectors provided with z/VSE are now introduced.

The Application Framework for e-business covers the most up-to-date security standards such as SSL, SET, and firewall, and supports:

- WebSphere Application Server
- VisualAge for Java
- DB2 Universal Database
- Lotus Domino
- CICS Transaction Gateway
- WebSphere MQ for z/VSE

The *z/VSE e-business Connectors* support the Application Framework for e-business, and provide you with the resources to extend your *core applications* to *e-business applications*. By doing so, you can protect and leverage existing core-application investments.

- *Core applications* (typically CICS, COBOL, VSAM) typically run on the z/VSE host, are critical to the company's operations, are expected to remain in production for many years to come, and usually represent an enormous investment of past resources.
- *e-business applications* are typically based upon common standards such as TCP/IP, HTML, XML, Secure Sockets Layer (SSL), Secure Electronic Transaction (SET) and so on, include both the server and client code written in Java, access relational data locally or remotely, and interface with end-users via a standard Web browser.

© Copyright IBM Corp. 2000, 2013

Connection Possibilities Using z/VSE e-business Connectors

These are the current z/VSE e-business Connectors:

- The *Java-based connector*, that allows you to implement a Java application that uses the VSE Beans class library to access all VSE file systems, submit jobs, issue console commands, and so on. An introduction is provided in "Overview of the Java-Based Connector" on page 23.
- The *DB2-based connector* that allows you to implement a Java application that uses DB2 Connect™ on the physical/logical middle-tier to access VSAM and DL/I data by calling a DB2 Stored Procedure on the z/VSE host. An introduction is provided in "Overview of the DB2-Based Connector" on page 83.
- The *VSE Script connector* that enables you to use VSE Script (batch) files containing statements written using the *VSE Script language* to obtain data from the z/VSE host. An introduction is provided in "Overview of the VSE Script Connector" on page 39.
- The *VSAM Redirector Connector* that handles requests to VSAM datasets and then either redirects the requests synchronously (via the VSAM Redirector Client), or stores changes made to VSAM datasets on the z/VSE host for further processing (via the VSAM Capture Exit). An introduction is provided in "Overview of the VSAM Redirector Connector" on page 45.
- The *Database Call Level Interface* that allows you to choose a database server (IBM DB2, Oracle, Microsoft SQL Server, MySQL, and so on) that runs on a platform other than z/VSE. An introduction is provided in "Overview of the Database Call Level Interface" on page 73.
- *SOAP connectivity* that provides a SOAP server and a SOAP client running in CICS on the basis of CICS Web Support. The SOAP server allows a Web service that is implemented as a CICS program, to be called from any kind of Web service client (Apache, AXIS, Microsoft .Net, C#, and so on). An introduction is provided in "Overview of z/VSE Support for Web Services and SOAP" on page 414.

Figure 1 on page 5 shows the connections you can make between:
- Web clients / the middle-tier, and
- the z/VSE host.

Notes:
1. The middle-tier can be either a "physical" middle-tier (such as an IBM pSeries® processor) or a "logical" middle-tier (*Linux on System z* on a System z mainframe).
2. The components of the Java-based connector are shown as shaded.
3. The DB2-based connector uses a connection between DB2 Connect on the physical/logical middle-tier and DB2 Server for VSE on the z/VSE host.

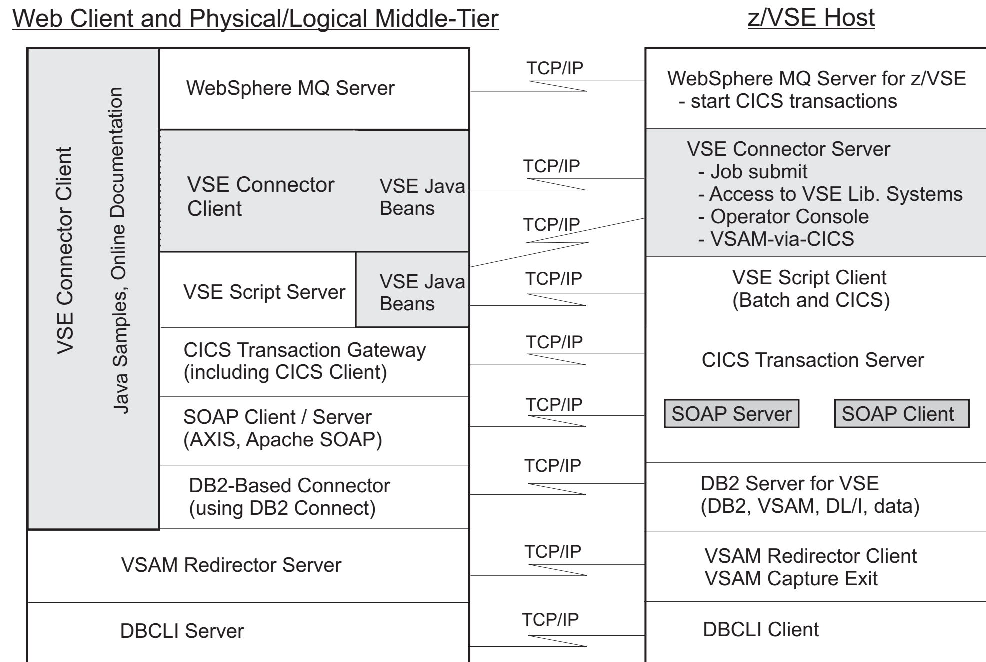


Figure 1. Overview of Connection Possibilities under z/VSE

Overview of CICS Connectivity

CICS connectivity in a *2-tier environment* enables CICS applications to be accessed using the CICS Web Support and 3270 Bridge (functions delivered with CICS Transaction Server for VSE/ESA).

CICS connectivity in a *3-tier environment* enables:

- A Java gateway application, that is usually stored on the physical/logical middle-tier, to communicate with CICS applications running in the CICS TS through the ECI (External Call Interface) or EPI (External Presentation Interface) provided by the CICS Universal Client.
  - The *ECI Interface* enables a non-CICS Client application to call a CICS program synchronously or asynchronously as a subroutine.
  - The *EPI Interface* enables a non-CICS Client application to act as a logical 3270 terminal and so control a CICS 3270 application.

  The CICS Universal Client communicates with the CICS TS via the APPC protocol.
- A CICS Java class library to be used for communication between the Java gateway application and a Java application (applet or servlet). The CICS Java class library also includes classes that provide an application programming interface (API):
  - Java programs can use the *JavaGateway* class to establish communication with the Gateway process, and this class uses Java's sockets protocol.
  - Java programs can use the:
    - *ECIRequest* class to specify the **ECI** calls that are flowed to the gateway.
    - *EPIRequest* class to specify **EPI** calls that are flowed to the gateway.
- A Web browser to be used as an emulator for a 3270 CICS application running on the CICS Transaction Server for VSE/ESA, via a Terminal Servlet.
- A set of Java EPI Beans to be used for creating Java front-ends for existing CICS 3270 applications, without any programming effort.

- The Simple Object Access Protocol (abbreviated to SOAP) to be used to send and receive information between CICS programs and other modules, over the Internet. For further information about SOAP, see Chapter 24, “Using SOAP for Inter-Program Communication,” on page 413.

Overview of WebSphere MQ Connectivity

WebSphere MQ connectivity in a 3-tier environment enables:

- Java applets to access WebSphere MQ queues on the z/VSE host. By exploiting the trigger facility provided by WebSphere MQ, you can start CICS applications that access CICS data and resources on the z/VSE host.
- a Web Client to *participate* in transactions, instead of simply providing and receiving information.

For further information about WebSphere MQ connectivity in a 3-tier environment, see Table 2 on page 15.

Overview of the IBM WebSphere Application Server

You implement the WebSphere Application Server on the physical/logical middle-tier of the *3-tier* Application Framework for e-business environment shown in Figure 3 on page 9. It is used together with a Web server (such as the IBM HTTP Server, or Apache server).

The WebSphere Application Server is fully compatible with industry standards such as Enterprise Java Beans (EJBs), eXtensible Markup Language (XML), and Common Object Request Broker Architecture (CORBA), and provides you with a solid framework for implementing your e-business applications.

Complementary programs include the:

1. WebSphere Studio, which is a powerful set of application development tools and facilities.
2. WebSphere Performance Pack.

In addition, you can use Tivoli's TME-10 network management together with the WebSphere Application Server.

Using the WebSphere Application Server, you can write applications (using Web development tools) for the physical/logical middle-tier that can access data and programs stored on the z/VSE host (CICS, DB2, and so on). These applications can be written to take advantage of the full benefits of Java and Internet technologies.

For an overview of where the WebSphere Application Server is used in 3-tier environments, see Figure 3 on page 9.

For more information about the WebSphere Application Server range, you might also refer to this Internet address:

www-4.ibm.com/software/webservers/appserv

Chapter 2. Overview of 2- and 3-Tier Environments

You can choose between 2-tier and/or 3-tier environments for the communication between Web clients, and the programs and data stored on the z/VSE host. These environments are illustrated in Figure 2 on page 8 and Figure 3 on page 9 respectively:

- In 2-tier environments, the Web client and the z/VSE host communicate directly with each other.
- In 3-tier environments, the Web client or non-Java client, and z/VSE host communicate with each other via an intermediate tier called the *middle-tier*. This middle-tier can be either a:
  - "physical" middle-tier (a processor such as an IBM xSeries® or pSeries running Linux, AIX®, or Windows).
  - "logical" middle-tier (*Linux on System z* on a System z mainframe).

Notes:

1. The 2-tier environment is *not* the typical environment under which you will develop your Java programs, since it is not:
   - part of the IBM *Application Framework for e-business* (it does not, for example, use the IBM WebSphere Application Server on the middle-tier).
   - secured by the state-of-the-art security services (firewall, and so on) provided by the IBM Application Framework for e-business.

   The 2-tier environment is generally suitable for *intranet* solutions only.
2. The Java-based connector is normally used in a 3-tier environment, but the DB2-based connector can *only* be used in a 3-tier environment. The 3-tier environment requires that the WebSphere Application Server is installed.

This chapter contains these main topics:

- "Overview of 2-Tier Environments"
- "Overview of 3-Tier Environments" on page 8
- "Where to Find Details of the Connectors" on page 9

Overview of 2-Tier Environments

In 2-tier environments as shown in Figure 2 on page 8:

1. The VSE Java Beans class library, which is part of the VSE Connector Client, must be accessible from each Java program running on a Web client, that communicates with the VSE Connector Server. This is achieved by copying the file **VSEConnector.jar** (which contains the VSE Java Beans) to each Web client on which your Java programs are to run.
2. The VSE Java Beans are used for establishing connections between the Java program running on the Web client and the VSE Connector Server running on the z/VSE host. Java applications or applets running on the Web clients can then use standard Web browsers to communicate directly with the VSE HTTP Server and VSE Connector Server running on the z/VSE host. When the work is complete, replies are sent from the VSE Connector Server to the Web client.
3. You can redirect VSAM requests to any Java-enabled platform using the VSAM Redirector Connector, and/or allow z/VSE applications to access a relational database on any suitable database server via the Database Call Level Interface.

© Copyright IBM Corp. 2000, 2013

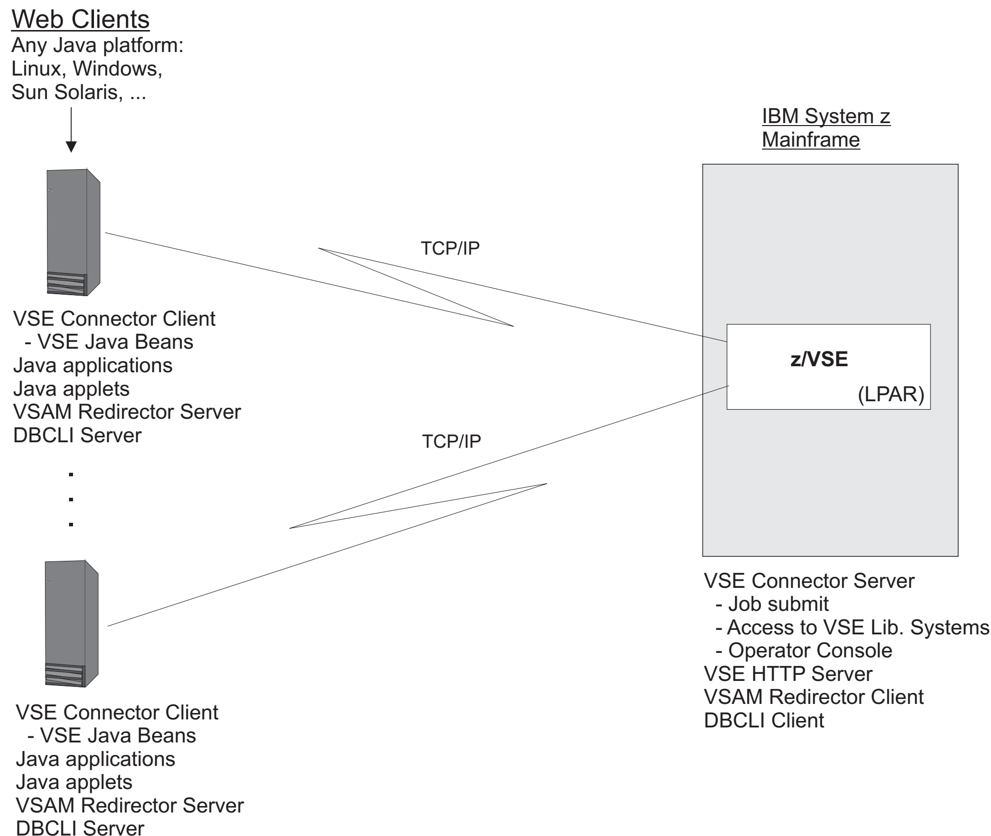


Figure 2. Overview of 2-Tier Environments and the Programs You Can Use

An example of a 2-tier implementation might be a Java application that directly communicates with the VSE Connector Server, to access POWER® data.

For a detailed description of how applets can be used in 2-tier environments, see Figure 102 on page 188

Overview of 3-Tier Environments

In your 3-tier environments as shown in Figure 3 on page 9, a *WebSphere Application Server* is the central “hub”. It runs either:

- under Linux, AIX, or Windows running (for example) on an IBM pSeries, xSeries, or Netfinity® processor (a *physical* middle-tier), or
- under *Linux on System z* on the mainframe (a *logical* middle-tier).

1. Either:
   - Web clients use standard Web browsers to communicate with an application that is based upon the WebSphere Application Server on the physical/logical middle-tier, or
   - Non-Java clients (for example a spreadsheet application running under Windows) use a VSE Script to communicate with the VSE Script Server running on the physical/logical middle-tier.
2. WebSphere applications, such as servlets, EJBs, or the VSE Script Server, use the VSE Java Beans to access VSE data or start an application which runs under z/VSE.
3. Replies are sent from the VSE Connector Server to the application running under the *WebSphere Application Server* (on the physical/logical middle-tier).

When the activities are completed, the application packages the data with other information, and sends a reply back to the Web client or the non-Java client.

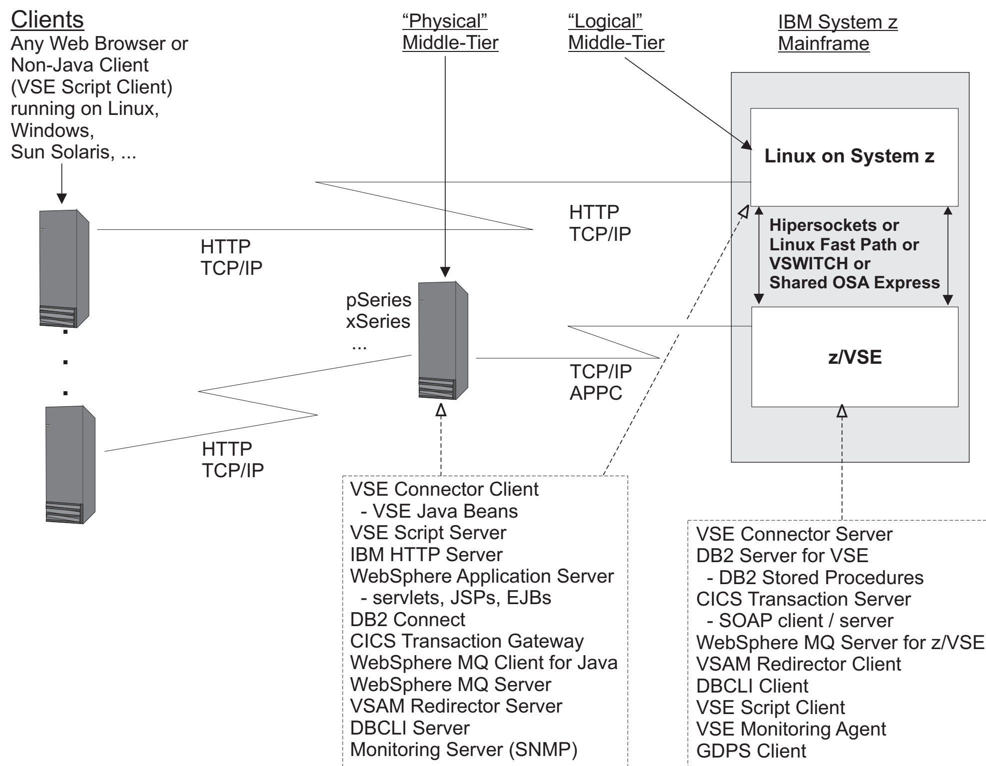


Figure 3. Overview of 3-Tier Environments and the Programs You Can Use

Note: In this publication, a *firewall* used in a 3-tier environment is *not* considered as a separate tier.

An example of a 3-tier implementation might be a Java servlet running on the Web Application Server (WebSphere) that provides access to VSAM data. The servlet might allow regular customers to enter orders over the Internet, using the required security layers.

Another (**non-Java**) example of a 3-tier implementation might be a Lotus® 1-2-3® application running under Windows (the VSE Script Client) that obtains VSAM data stored on the z/VSE host, via the VSE Script Server running on the physical/logical middle-tier. VSE Script connector is the term used to refer to both the VSE Script Client and the VSE Script Server.

Where to Find Details of the Connectors

For detailed information of how:

- Applets can be used, see Chapter 17, "Using Java Applets to Access Data," on page 187.
- Servlets can be used, see Chapter 18, "Using Java Servlets to Access Data," on page 227.
- Java Server Pages (JSPs) can be used, see Chapter 19, "Using Java Server Pages to Access Data," on page 245.

- Enterprise Java Beans (EJBs) can be used, see Chapter 20, "Using EJBs to Represent Data," on page 249.
- DB2 Stored Procedures can be used to access VSE/VSAM data, see "Using DB2 Stored Procedures to Access VSAM Data" on page 395.
- DB2 Stored Procedures can be used to access DL/I data, see "Using DB2 Stored Procedures to Access DL/I Data" on page 402.
- The VSAM Redirector Connector can be used to redirect requests for VSAM data, see Chapter 8, "Installing the VSAM Redirector Connector," on page 45.
- The Database Call Level Interface can be used to allow z/VSE applications to access a relational database on any suitable database server, see Chapter 9, "Installing the Database Call Level Interface," on page 73.
- The Simple Object Access Protocol (abbreviated to SOAP) can be used to send and receive information between CICS programs and other modules over the Internet, see Chapter 24, "Using SOAP for Inter-Program Communication," on page 413.
- The VSE Script connector can be used for non-Java access to functions and data stored on the z/VSE host, see Chapter 25, "Using the VSE Script Connector for Non-Java Access," on page 437.
- The VSE Monitoring Agent can be used to send SNMP Version 1 "traps" to SNMP monitors from batch jobs, COBOL, C, and CICS programs, see Chapter 13, "Collecting Data via the VSE Monitoring Agent," on page 115.
- The GDPS Client can be used to enable availability data to be collected from z/VSE systems and sent to a central *GDPS K-System* running under z/OS, see Chapter 14, "Using GDPS Support for High Availability," on page 129.

Part 2. INSTALLATION & CUSTOMIZATION

Chapter 3. Choosing the Connectivity You Require . . . 13
Connectivity Possibilities in 2-Tier Environments . . 13
Connectivity Possibilities in 3-Tier Environments . . 14

Chapter 4. Installing the Common Prerequisite Programs . . . 19
Configuring and Activating TCP/IP for VSE/ESA 19
Configuring and Activating the VSE HTTP Server 19
Installing and Configuring Java . . . 19
Downloading the Java Base Code . . . 20
Deciding Which Java Package to Install . . . 20
Installing the IBM HTTP Server . . . 20
Installing the WebSphere Application Server . . . 21
Installing the WebSphere Application Server on Linux on System z . . . 21
Installing the WebSphere Application Server on z/OS . . . 21
Installing the WebSphere Application Server on Other Platforms . . . 21

Chapter 5. Installing and Operating the Java-Based Connector . . . 23
Overview of the Java-Based Connector . . . 23
Overview of the VSE Connector Client . . . 23
Overview of the VSE Connector Server . . . 24
Installing the VSE Connector Client . . . 24
Obtaining a Copy of the VSE Connector Client 25
Performing the VSE Connector Client Installation 26
Using the Online Documentation Options . . . 26
Configuring for WebSphere Support . . . 27
Uninstalling the VSE Connector Client . . . 27
Configuring the VSE Connector Server . . . 27
Job SKVCSSTJ – Startup Job . . . 28
Job SKVCSCAT – Catalog Members . . . 28
VSE Library Member SKVCSCFG – General Settings . . . 29
VSE Library Member SKVCSLIB – Specify Libraries to Be Accessed . . . 31
VSE Library Member SKVCSPLG – Specify Plugins to Be Loaded . . . 31
VSE Library Member SKVCSUSR – Specify Logon Access . . . 32
VSE Library Member SKVCSSSL – Configure for SSL . . . 32
Configuring the Date-Format for the VSE Connector Server . . . 33
Starting the VSE Connector Server . . . 34
Testing the Communication Between VSE Connector Client and Connector Server . . . 34
Obtaining a List of VSE Connector Server Commands . . . 35
Entering a Command for the VSE Connector Server 35
Maintaining Security Using the VSE Connector Server . . . 36

Chapter 6. Configuring DL/I for Access Via VSE Java Beans . . . 37
Host Installation Activities That Must Be Already Completed . . . 37
Step 1: Skeleton SKDLISMP – Define Sample Database . . . 37
Step 2: Customize CICS TS . . . 38

Chapter 7. Installing the VSE Script Connector 39
Overview of the VSE Script Connector . . . 39
Step 1: Download the Install-File and Perform the Installation . . . 40
Step 1.1: Obtain a Copy of the VSE Script Server 40
Step 1.2: Perform the Installation of the VSE Script Server . . . 41
Step 2: Configure the VSEScriptServer Properties File . . . 42
Step 3: Configure the Connections Properties File . . 43

Chapter 8. Installing the VSAM Redirector Connector . . . 45
Overview of the VSAM Redirector Connector . . . 45
VSAM Integration Considerations . . . 49
Configuring the VSAM Redirector Client / VSAM Capture Exit . . . 49
Step 1: Enable the VSAM Redirector Client / VSAM Capture Exit on z/VSE . . . 50
Step 2: Decide Upon Your Redirection Mode . . 51
Redirection Modes Available for the VSAM Redirector Client . . . 51
Redirection Modes Available for the VSAM Capture Exit . . . 55
Understanding the Layout of Delta Records/Messages . . . 56
Using a VSAM Delta Cluster . . . 56
Using WebSphere MQ . . . 57
Step 3 (Optional): Transfer Your VSAM Data . . 58
Step 4: Create the Configuration Phase . . . 58
Mandatory Parameters . . . 58
Optional Parameters . . . 59
Parameters When Using the IESREDIR Exit. . 59
Parameters When OWNER=VSAM . . . 60
Parameters When Using the IESVSCAP Exit 60
Parameters When MODE=JOURNALING or MODE=CUMULATIVE . . . 61
Parameters When MODE=MQSERVER and MODE=MQCLIENT . . . 61
Parameters When MODE=MQCLIENT . . . 61
Example of Job to Create the Configuration Phase . . . 62
Installing the VSAM Redirector Server . . . 63
Step 1: Download the Install-File and Perform the Installation . . . 63
Step 1.1: Obtain a Copy of the VSAM Redirector Server . . . 63

© Copyright IBM Corp. 2000, 2013

Step 1.2: Perform the Installation of the VSAM Redirector Server 64
Step 2: Configure the Properties File 65
Step 3: Implement a VSAM Redirector Handler 65
Coding VSAM Logic and Parameters 65
Calling a VSAM Redirector Handler 66
Error Reporting 66
Datatype Conversions 67
Getting a Map Dynamically Into Your Redirector Handler 67
Using the IBM-Provided VSAM Redirector Handlers 67
Using the IBM-Provided VSAM Redirector Loaders 69
Description of the RedirLoader Redirector Loader 69
Description of the MQLoader Redirector Loader 69
Description of the DeltaLoader Redirector Loader 70
IBM-Supplied Example of DB2-Related Handler . . 70

Chapter 9. Installing the Database Call Level Interface 73
Overview of the Database Call Level Interface . . . 73
Overview of Connection Pooling 74
Prerequisites for Using the Database Call Level Interface 74
Installing the DBCLI Server 75
Obtaining a Copy of the DBCLI Server 75
Performing the DBCLI Server Installation . . . 76
Uninstalling the DBCLI Server 76
Setting Up and Configuring the DBCLI Server. . . 76
DatabaseCliServer.cfg 77
JdbcAliases.cfg 78
JdbcDriver.cfg 78
Configuring and Starting/Stopping the Connection Pool Manager 78
Defining CICS Programs/Transactions and OME for Connection Pooling 79
Starting the Connection Pool Manager 79
Stopping the Connection Pool Manager 80
Querying the Connection Pool 81
DBCLI Server Commands 81

Chapter 10. Customizing the DB2-Based Connector 83
Overview of the DB2-Based Connector 83
Host Installation Activities That Must Be Already Completed 84
Step 1: Customize CICS TS 84
Step 2: Customize TCP/IP 85
Step 3: Customize DB2 and Define Sample Database 85
Step 3.1: Define User Catalog 85
Step 3.2: Catalog New ARISIVAR.Z 86
Step 3.3: Job Manager for Preparation / Installation Steps 87
Step 3.4: Activate DRDA Server Support. . . . 87
Step 3.5: Startup Job for Stored Procedure Server 87
Step 3.6: Prepare DB2 Sample Database 88
Step 3.7: Install DB2 Sample Database 89
Step 4: Set Up for DRDA Support 93
Step 5: Set Up Stored Procedure Server and Define to DB2 93
Step 5.1: Set Up the Stored Procedure Server . . 93
Step 5.2: Define Stored Procedure Server to DB2 94
Step 6: Set Up for Stored Procedures 94
Step 7: Customize the DB2-Based Connector for VSAM Data Access 95
Step 8: Customize the DB2-Based Connector for DL/I Data Access 95
Step 9: Start DB2, and Start Stored Procedure Server 96
Step 10: Install DB2 Connect and Establish Client-Host Connection 96

Chapter 11. Configuring the VSAM-Via-CICS Service 99
Configuring the IBM-Supplied CICS System . . . 99
Configuring a Further CICS System for VSAM-Via-CICS 100
How the VSAM-Via-CICS Service Works 101
CICS Transactions for Use with VSAM-Via-CICS 101

Chapter 12. Mapping VSE/VSAM Data to a Relational Structure 103
Introduction to Mapping VSE/VSAM Data . . . 103
How VSAM Maps Are Structured 104
How Maps Are Stored on the z/VSE host 104
Defining a Map Using RECMAP 105
Defining a Map Using the Sample Applet 105
Defining a Map Using a Java Application 106
Defining a Map Using the VSAM MapTool . . . 112

Chapter 3. Choosing the Connectivity You Require

For 2-tier and 3-tier environments, this chapter describes the types of connectivity you can establish using the Java-based connector, the DB2-based connector, the VSAM Redirector connector, the VSE Script connector, CICS (including the SOAP server and SOAP client), and WebSphere MQ.

This chapter contains these main topics:
- "Connectivity Possibilities in 2-Tier Environments"
- "Connectivity Possibilities in 3-Tier Environments" on page 14

Connectivity Possibilities in 2-Tier Environments

For 2-tier environments, you can use:
- The Java-based connector
- The CICS *Web Support* feature
- The VSAM Redirector connector
- The Database Call Level Interface

to carry out the functions described in Table 1.

Table 1. Connectivity Possibilities in 2-Tier Environments

Connectivity	Java Applications	Servlets / Java Server Pages	Applets	Enterprise Java Beans
Java-Based Connector	Implement a Java application on the client, that uses the VSE Beans class library to access all VSE file systems (see **Note** below), submit jobs, issue console commands, etc.	Not possible	Download an applet from VSE to your Web browser. Then use the VSE Beans class library to access all VSE file systems (see **Note** below), submit jobs, issue console commands, and so on. The VSE Beans classes can be put into the same JAR file together with the applet code.	Not possible
CICS	Use the *Web Support* feature provided by the CICS Transaction Server for VSE/ESA, to access CICS transactions directly from a Web browser. The CICS transactions generate the required Web pages, which are then displayed by the Web browser. However, since the use of the Web Support feature in a z/VSE environment is generally outside the scope of this publication, for details you should refer to the *CICS Transaction Server for VSE/ESA, Internet Guide*, SC34-5765.			
VSE Web Services	Provides a SOAP server and a SOAP client running in CICS on the basis of CICS Web Support. The server allows a Web service that is implemented as a CICS program, to be called from any kind of Web service client (Apache, AXIS, Microsoft .Net, C#, and so on). The client can call any Web service on, e.g., a WebSphere.			

© Copyright IBM Corp. 2000, 2013

Table 1. Connectivity Possibilities in 2-Tier Environments (continued)

<table>
<tr><th>Connectivity</th><th>Java Applications</th><th>Servlets / Java Server Pages</th><th>Applets</th><th>Enterprise Java Beans</th></tr>
<tr><td>VSAM Redirector connector</td><td colspan="4">The VSAM Redirector connector handles requests to VSAM datasets and then either:
• redirects the requests synchronously (via the VSAM Redirector Client).
• stores changes made to VSAM datasets on the z/VSE host for further processing (via the VSAM Capture Exit).
Your source programs do not need to be changed.</td></tr>
<tr><td>Database Call Level Interface</td><td colspan="4">The Database Call Level Interface can be used to allow z/VSE applications to access a relational database (IBM DB2, Oracle, Microsoft SQL Server, MySQL, and so on) on any suitable database server.</td></tr>
<tr><td>SOAP connectivity</td><td colspan="4">SOAP is a standard, XML-based, industry-wide protocol that allows applications to exchange information over the Internet via HTTP. SOAP combines the benefits of both XML and HTTP into one standard application protocol. As a result, you can send and receive information to/from various platforms. z/VSE supports the SOAP protocol and therefore allows you to implement Web services.</td></tr>
</table>

Note: The term *VSE file systems* includes VSE/VSAM, VSE/POWER, VSE/Librarian, and VSE/ICCF.

Connectivity Possibilities in 3-Tier Environments

For 3-tier environments, you can use:

- The Java-based connector
- The DB2-based connector
- The VSE Script connector
- VSAM Redirector connector
- Database Call Level Interface
- CICS connectivity, including the Simple Object Access Protocol (abbreviated to SOAP)
- WebSphere MQ connectivity

to carry out the functions described in Table 2 on page 15.

Table 2. Connectivity Possibilities in 3-Tier Environments

Connectivity	Java Applications	Servlets / Java Server Pages	Applets	Enterprise Java Beans
Java-based connector	Not applicable.	Implement a servlet using the VSE Beans class library to access the z/VSE host data and display it through HTML pages. Refer to the online documentation (described on page "Using the Online Documentation Options" on page 26) for details.	Download an applet from the middle tier's Web server and use the *VSEAppletServer* (see page "How the VSEAppletServer Is Used" on page 191 for a description) to connect to a remote z/VSE host to access VSE file systems (see **Note** at end of this table). Refer to the online documentation (described on page "Using the Online Documentation Options" on page 26) for details.	Write an EJB that uses the VSE Java Beans class library to communicate with the z/VSE host and access your database on the z/VSE host. Normally, this will be either DB2 data or VSE/VSAM data. Refer to the online documentation (described on page "Using the Online Documentation Options" on page 26) for details.
DB2-based connector	Implement a Java application on the Web client, that uses DB2 Connect on the physical/logical middle-tier to access VSAM and DL/I data by calling a DB2 Stored Procedure on the z/VSE host.	Write a servlet or JSP that uses DB2 Connect to access VSAM and DL/I data by calling a DB2 Stored Procedure on the z/VSE host. The servlet or JSP can be accessed from any Web client through WebSphere.	Download an applet from the physical/logical middle-tier's web server. The applet connects to DB2 Connect on the physical/logical middle-tier, which accesses VSAM and DL/I data by calling a DB2 Stored Procedure on the z/VSE host.	Write an EJB that uses DB2 Connect to access VSAM and DL/I data by calling a DB2 Stored Procedure on the z/VSE host.
VSE Script connector	Implement a non-Java program to call a VSE Script to access VSE functions and data. For example, write a Visual Basic™ script within an Office product such as Lotus 1-2-3, to include VSE data in a spreadsheet or document.	Not applicable.	Not applicable.	Not applicable.

Table 2. Connectivity Possibilities in 3-Tier Environments (continued)

Connectivity	Java Applications	Servlets / Java Server Pages	Applets	Enterprise Java Beans
VSAM Redirector connector	The VSAM Redirector connector handles requests to VSAM datasets and then either: • redirects the requests synchronously (via the VSAM Redirector Client). • stores changes made to VSAM datasets on the z/VSE host for further processing (via the VSAM Capture Exit). Your source programs do not need to be changed.	Not applicable.	Not applicable.	Not applicable.
Database Call Level Interface	The Database Call Level Interface can be used to allow z/VSE applications to access a relational database (IBM DB2, Oracle, Microsoft SQL Server, MySQL, and so on) on any suitable database server.	Not applicable.	Not applicable.	Not applicable.
CICS	Implement a Java application on the Web client, that connects to the CICS Transaction Gateway on the physical/logical middle-tier. Use the ECI/EPI interface to communicate with the CICS TS.	Write a servlet or JSP that uses the CICS ECI/EPI interface to communicate with the CICS TS. The servlet or JSP can be accessed from any Web client through WebSphere.	Download an applet from the physical/logical middle-tier's Web server. The applet connects to the CICS Transaction Gateway on the physical/logical middle-tier, which then communicates with the CICS TS.	Write an EJB that uses the CICS client's ECI/EPI interface to communicate with the CICS TS.

Table 2. Connectivity Possibilities in 3-Tier Environments (continued)

Connectivity	Java Applications	Servlets / Java Server Pages	Applets	Enterprise Java Beans
WebSphere MQ	Implement a Java application on the Web client, that uses the WebSphere MQ Client for Java to connect to (for example) the WebSphere MQ Server for Windows on the physical/logical middle-tier. The WebSphere MQ Server for Windows (for example) on the physical/logical middle-tier in turn connects to the WebSphere MQ Server for z/VSE® on the z/VSE host. You can then start any CICS transaction on the z/VSE host from a Java application on the Web client.	Write a servlet or Java Server Page (JSP) that uses the WebSphere MQ Client for Java to start any CICS transaction on the z/VSE host. The servlet or JSP can be accessed from any Web client through WebSphere.	Download an applet from the physical/logical middle-tier's Web server and use a "router" to connect to a remote z/VSE host to access VSE based data.	Write an Enterprise Java Bean (EJB) that uses the WebSphere MQ Client for Java to communicate with the z/VSE host and access a database there.

Note: The term *VSE file systems* includes VSE/VSAM, VSE/POWER, VSE/Librarian, and VSE/ICCF.

Chapter 4. Installing the Common Prerequisite Programs

This chapter describes the activities you must perform and which are independent of your choice of connectors.

It consists of these main topics:

- "Configuring and Activating TCP/IP for VSE/ESA"
- "Configuring and Activating the VSE HTTP Server"
- "Installing and Configuring Java"
- "Installing the IBM HTTP Server" on page 20
- "Installing the WebSphere Application Server" on page 21

Configuring and Activating TCP/IP for VSE/ESA

You require TCP/IP for VSE/ESA in order to use the:

- Java-based connector
- DB2-based connector
- VSAM Redirector connector
- VSE Script connector

TCP/IP for VSE/ESA is supplied with z/VSE, but requires a key to be activated, which you must purchase. For details on how to activate and configure TCP/IP for VSE/ESA, refer to the *z/VSE TCP/IP Support*.

Note: Where you can use VTAM / APPC instead of TCP/IP, is noted in this publication.

Configuring and Activating the VSE HTTP Server

The TCP/IP for VSE/ESA *Application Pak* includes the VSE HTTP Server. You should therefore refer to the *TCP/IP for VSE/ESA Installation Guide*, SC33-6741, for details of how to configure and activate the VSE HTTP Server. This publication is available as a *PDF file only*. You can obtain this publication from the *VSE Collection* online library. You can obtain the VSE Collection online library as either a:

- CD-ROM (number SK2T-0060).
- DVD (number SK3T-8348).

Installing and Configuring Java

Java is a programming language in which *bytecode* is created from Java source files. This bytecode is stored in Java *class files.* These class files are read and executed by the Java interpreter (for Windows and OS/2, this is the program **java.exe**).

As stated previously, the z/VSE Java-based connector consists of the VSE Connector Client and the VSE Connector Server. To develop Web applications that use the *VSE Connector Client*, you must install the Java Development Kit (JDK) 1.4 or later on your development platform.

© Copyright IBM Corp. 2000, 2013

Notes:

1. You can install different versions of Java on the same physical/logical middle-tier platform, but to do so you must ensure that the correct paths are set for the Java version that is to run.
2. To use some of the samples supplied with the Java-based connector, you require the Java *Swing* classes. These are supplied when you install the VSE Connector Client (described in "Installing the VSE Connector Client" on page 24).

Downloading the Java Base Code

You can download Java-related code from these web sites:

- `http://www.ibm.com/developerworks/java/jdk/` from which you can download, for example, the Java Development Kits (JDKs) for AIX, OS/2, z/OS® (Unix services), OS/400®, z/VM®, Linux, and Windows.
- `http://www.ibm.com/developerworks/java/` from which you are given access to various libraries containing both tools and documentation.
- `http://www.oracle.com/technetwork/java/` from which you can download, for example, the Java 2 Platform Standard Edition for Windows, Linux, or Solaris platforms.

Deciding Which Java Package to Install

You can install Java in one of two ways on your physical/logical middle-tier server:

- As a Java Development Kit (JDK) installation, in which you use **java.exe** to run your Java programs. The JDK contains the run-time environment, together with tools and facilities such as debugger, compiler, and so on. You will require the JDK if you are developing your own Java programs.
- As a Java Runtime Environment (JRE) installation, in which the JRE contains only the runtime library required to run Java applications (development tools are not included). Here, you use **jre.exe** to run your Java programs.

Installing the IBM HTTP Server

On your physical/logical middle-tier, you must install a *Web Server* which can be, for example, the:

- IBM HTTP Server
- Lotus Domino® Go Webserver
- Apache Server

The IBM HTTP Server is part of the WebSphere Application Server package. During the installation of the WebSphere Application Server, you are asked if you wish to:

- install the IBM HTTP Server as your Web Server.
- use another Web Server (for example, the Apache Server).

For details of how to install the IBM HTTP Server on your physical/logical middle-tier, refer to the installation instructions provided with the WebSphere Application Server.

Installing the WebSphere Application Server

On your *middle-tier*, which can be either a "physical" middle-tier or a "logical" middle-tier (see "Overview of 3-Tier Environments" on page 8), you must install an *Application Server,* which can be the IBM WebSphere Application Server, or any other vendor's application server.

The WebSphere Application Server is supplied in these editions:

- The Standard Edition, which supports servlets and Java Server Pages (JSPs).
- The Advanced Edition or the Enterprise Edition, which both support servlets, JSPs, and Enterprise Java Beans (EJBs).

Because the installation of the WebSphere Application Server on various platforms is both complex and subject to change, this topic only provides documentation references and/or a description of the general installation steps.

Installing the WebSphere Application Server on Linux on System z

Refer to the relevant IBM publications for information on how to install the WebSphere Application Server and Supporting Products on the Linux on System z platform on the mainframe.

Installing the WebSphere Application Server on z/OS

Refer to the relevant IBM publications for information on how to install the WebSphere Application Server and Supporting Products on the z/OS platform on the mainframe.

Installing the WebSphere Application Server on Other Platforms

These are the *general* steps you might follow to install the WebSphere Application Server on platforms such as Windows, Linux, AIX or Sun Solaris:

1. Make sure you meet the hardware and software prerequisites for the platform and configuration you plan to install.
2. Decide on which of the installation steps you need to follow. For example, if your system does not already have the WebSphere Application Server, IBM Java Development Kit, IBM HTTP Server, and DB2 Universal Database installed, you will follow a different procedure than if your system already has these products installed.
3. Install the IBM Java Development Kit by:
   a. running an **exec** file
   b. following the instructions displayed on each window.
4. Install the IBM HTTP Server (or another Web Server, such as Apache).
5. Install the IBM DB2 Universal Database (or another database system, such as Oracle).
6. Test the installation of the prerequisite products for your configuration.
7. Install the WebSphere Application Server on the platform you have chosen.
8. Test the installation of the WebSphere Application Server, together with the prerequisite products.

However, your main source of information for installing the WebSphere Application Server on platforms such as Windows, Linux, AIX, or Sun Solaris, should be the IBM Web pages that are kept up-to-date. You can find the IBM Web pages at:

```
http://www.ibm.com/software/webservers/appserv/library.html
```

Chapter 5. Installing and Operating the Java-Based Connector

This chapter describes how you install and operate the Java-Based Connector.

It consists of these main topics:

- "Overview of the Java-Based Connector"
- "Installing the VSE Connector Client" on page 24
- "Uninstalling the VSE Connector Client" on page 27
- "Configuring the VSE Connector Server" on page 27
- "Configuring the Date-Format for the VSE Connector Server" on page 33
- "Starting the VSE Connector Server" on page 34
- "Testing the Communication Between VSE Connector Client and Connector Server" on page 34
- "Obtaining a List of VSE Connector Server Commands" on page 35
- "Entering a Command for the VSE Connector Server" on page 35
- "Maintaining Security Using the VSE Connector Server" on page 36

Overview of the Java-Based Connector

The *Java-based connector* consists of a client-part (the *VSE Connector Client*), and a server-part (the *VSE Connector Server*).

Overview of the VSE Connector Client

You can install the VSE Connector Client on most Java-enabled platforms. It consists of these parts:

- a main file **VSEConnector.jar** which contains the VSE Java Beans class library. The VSE Java Beans provide a Java programming interface for communicating with VSE/VSAM, VSE/Librarian, VSE/POWER, VSE/ICCF, and Operator Console, on the z/VSE host.
- three additional jar files (**ibmjsse.jar**, **cci.jar**, and **ibmpkcs.jar**).
- a set of samples, including Java source code, that show you how to write Java programs that are based upon the use of VSE Java Beans.
- online documentation (a set of HTML pages) describing the various concepts and samples.

To develop your Web applications, you will probably use all the three components listed above. However, the completed Web applications require only the VSE Java Beans class library at run-time. Therefore, when your Web applications are ready for the production environment (the end-user environment), you install only the **VSEConnector.jar** file (which contains the VSE Java Beans class library) on the:

- Physical/logical middle-tier (for 3-tier environments)
- Web clients (for 2-tier environments).

The difference between 2-tier and 3-tier environment is explained in Chapter 2, "Overview of 2- and 3-Tier Environments," on page 7.

© Copyright IBM Corp. 2000, 2013

Overview of the VSE Connector Server

The VSE Connector Server is installed on the z/VSE host. It is a batch application that runs by default in dynamic class R. After you have configured the VSE Connector Server, it must simply be started in order to become operational. It then provides a TCP/IP *socket listener* which can handle multiple clients.

Java programs running on Web clients or the physical/logical middle-tier use the VSE Java Beans to build connections to the VSE Connector Server running on the z/VSE host.

To start the VSE Connector Server, you use the job **STARTVCS**, which is placed in the POWER reader queue during the installation of the z/VSE base. When started, the VSE Connector Server listens to incoming TCP/IP traffic on port 2893 by default. For details of how to start the VSE Connector Server, see "Starting the VSE Connector Server" on page 34.

The VSE Connector Server is pre-configured for your use, and should require no major configuration effort by yourself. However, you can modify various configuration members to specify:

- The VSE/AF libraries that can be accessed by the VSE Connector Server. You may extend or restrict this list according to your needs.
- Plugins to be loaded at VSE Connector Server startup. You can extend the Java-based connector by specifying your own host-side plugins, as described in Chapter 21, "Extending the Java-Based Connector," on page 269.
- Which users or groups of users are allowed to logon to the VSE Connector Server.

For an overview of where the VSE Connector Client and VSE Connector Server are used in 2-tier and 3-tier environments, see Figure 2 on page 8 and Figure 3 on page 9.

Installing the VSE Connector Client

This topic describes how you install the VSE Connector Client on the physical/logical middle-tier server of a *3-tier environment*.

The 3-tier environment is described in "Overview of 3-Tier Environments" on page 8.

If you plan to implement a 2-tier environment using applets and Java applications that run on the Web clients, you must copy the VSE Java Beans part of the VSE Connector Client (file **VSEConnector.jar**) to each Web client on which the applets and Java applications are to run. The VSE Java Beans are required in order to establish connections between Web clients and the VSE Connector Server running on the z/VSE host.

The VSE Connector Client is included in the VSE Central Functions and consists of one file **iesincon.w**.

Obtaining a Copy of the VSE Connector Client

Note: Before you begin, you must already have installed the Java Development Kit (JDK) 1.4 or later on the development platform where you plan to install the VSE Connector Client. If you do not have JDK 1.4 or later installed, refer to "Installing and Configuring Java" on page 19 for details of how to install it.

To obtain a copy of the VSE Connector Client, you must decide if you wish to obtain it:
- from the Internet.
- by installing the VSE Connectors Workstation Code component from the Extended Base Tape.

To obtain the client *from the Internet*, you should:
1. Start your Web browser and go to the URL

   `http://www.ibm.com/systems/z/os/zvse/downloads/`
2. From within the VSE Connector Client section, download the file **vsecon*nnn*.zip** to the directory where you wish to install the VSE Connector Client. **Note:** *nnn* refers to the current VSE version (for example, **vsecon510.zip**).

To obtain the client *by installing the VSE Connectors Workstation Code component*, you should:
1. Install the VSE Connectors Workstation Code component from the Extended Base Tape. After you have installed this component, the VSE Connector Client W-book **iesincon.w** will be stored in z/VSE sublibrary PRD2.PROD.
2. Use the FTP (file transfer program) utility of TCP/IP for VSE/ESA to download **iesincon.w** to the directory where you wish to install the VSE Connector Client.

Notes:
1. You must download **iesincon.w** in *binary*.
2. Make sure that Unix mode is ***turned off***. Otherwise **iesincon.w** will be downloaded in ASCII mode, even when you specify *binary*. *Unix mode* is one parameter of your VSE FTP daemon. Some FTP clients might *force* Unix mode to be turned on! The example below shows how a successful transfer of **iesincon.w** was made using a batch FTP client. The place where the UNIX mode is set, is shown as bold.

```
c:\temp>ftp 9.164.155.2
Connected to 9.164.155.2.
220-TCP/IP for VSE -- Version 01.05.F  -- FTP Daemon
    Copyright (c) 1995,2006 Connectivity Systems Incorporated
220 Service ready for new user.
User (9.164.155.2:(none)): sysa
331 User name okay, need password.
Password:
230 User logged in, proceed.
ftp> cd prd2
250 Requested file action okay, completed.
ftp> cd prod
250 Requested file action okay, completed.
ftp> binary
200 Command okay.
ftp> get iesincon.w
200 Command okay.
150-File: PRD2.PROD.IESINCON.W
    Type: Binary Recfm: FB Lrecl:    80 Blksize:      80
    CC=ON  UNIX=OFF RECLF=OFF TRCC=OFF CRLF=ON  NAT=NO
```

```
150 File status okay; about to open data connection
226-Bytes sent:       4,756,400
    Records sent:        59,455
    Transfer Seconds:        16.52 (    290K/Sec)
    File I/O Seconds:         3.94 (  1,548K/Sec)
226 Closing data connection.
4756400 bytes received in 17,12 seconds (277,91 Kbytes/sec)
ftp> bye
221 Service closing control connection.
c:\temp>ren iesincon.w vsecon.zip
```

Performing the VSE Connector Client Installation

To perform the installation of the VSE Connector Client, you must:

1. Unzip the file **vsecon.zip**, which contains these files:
   - setup.jar (contains the VSE Connector Client code)
   - setup.bat (an install batch file for Windows XP/Vista/7)
   - setup.cmd (an install batch file for Windows NT)
   - setup.sh (an install script for Linux/Unix)
2. Start the batch file (by double-clicking the file) that is applicable to your operating-system platform.
3. The installation process now begins, and you are guided through various installation menus.
4. To access the HTML-based documentation, you can now use your Web browser to open the file **VSEConnectors.html**.

Using the Online Documentation Options

After installing the VSE Connector Client, you can then access all information (including samples and source code) *via the online documentation*.

The main window of the Online Documentation contains links (on the left of this window) to:

- The Java-based connector and DB2-based connector, but also to the CICS connectivity and WebSphere MQ connectivity setups. Here you find information and examples for these connectors and setups.
- Explanations of the concepts, such as Java applets, servlets, Java Server Pages (JSPs), Enterprise Java Beans (EJBs), and JDBC. Here you find information and examples related to these topics.
- Online reference (under **All classes**), including references to information about how you can extend the VSE Java Beans by writing your own *Plugins*.
- Samples that you can run directly from a Web browser. To run these samples, you must have TCP/IP for VSE/ESA and the VSE Connector Server running on your z/VSE host.

  Note: Before you can run any of the samples, or write your own Java programs that use the VSE Java Beans, you must include the **VSEConnector.jar** file in your local CLASSPATH variable.

Configuring for WebSphere Support

If you plan to implement the IBM WebSphere Application Server on your physical/logical middle-tier, you should now complete the steps contained in the online documentation which describe how to configure your physical/logical middle-tier for WebSphere. To find this information, click **Further Information** in the Online Documentation window (described in "Using the Online Documentation Options" on page 26), then select the appropriate version of WebSphere for your installation.

Now follow the steps described in the online documentation.

Uninstalling the VSE Connector Client

You have two possible methods to uninstall the VSE Connector Client:

- By using the normal *Windows* **Add or Remove Programs** function, which is selected via the Windows Control Panel.
- By manually running the VSE Connector Client uninstall program. You can find this program in the **_uninst** subdirectory that is located within the directory where you installed the VSE Connector Client. *This option is available for all platforms*.

Configuring the VSE Connector Server

The VSE Connector Server is an application that runs in batch in one of your z/VSE partitions, and which implements a TCP/IP connection.

A description of the VSE Connector Server is provided in "Overview of the VSE Connector Server" on page 24.

This topic describes the jobs you use to configure your VSE Connector Server:

SKVCSSTJ
A skeleton startup job.

SKVCSCAT
A job to catalog the VSE Connector Server's configuration members. These are the skeletons contained within job SKVCSCAT:

SKVCSCFG
A VSE library member in which you specify the general settings for the VSE Connector Server

SKVCSLIB
A VSE library member in which you specify the VSE libraries that can be accessed by the VSE Connector Server

SKVCSPLG
A VSE library member in which you specify the server plugins to be loaded during startup of the VSE Connector Server.

SKVCSUSR
A VSE library member in which you specify the users, or groups of users, who can logon to the VSE Connector Server.

SKVCSSSL
A VSE library member in which you configure the VSE Connector Server for Secure Sockets Layer (SSL) security.

Job SKVCSSTJ – Startup Job

You can find the skeleton job SKVCSSTJ in the VSE/ICCF library 59.

You use SKVCSSTJ to place a startup job (for starting the VSE Connector Server) in the VSE/POWER reader queue. SKVCSSTJ is also available as a Z book, which is loaded into the POWER reader queue during initial installation of z/VSE, or during a cold startup of z/VSE.

```
* $$ JOB JNM=CATSTVCS,DISP=D,CLASS=0
// JOB CATSTVCS       CATALOG STARTVCS AND LDVCS, LOAD STARTVCS
// EXEC LIBR,PARM='MSHP'
ACC S=IJSYSRS.SYSLIB
CATALOG  STARTVCS.Z        REPLACE=YES
$$$$ JOB JNM=STARTVCS,DISP=L,CLASS=R
$$$$ LST CLASS=A,DISP=D
// JOB STARTVCS START UP VSE CONNECTOR SERVER
// ID USER=VCSRV
*   WAITING FOR TCP/IP TO COME UP
// EXEC REXX=IESWAITR,PARM='TCPIP00'
// LIBDEF *,SEARCH=(PRD2.CONFIG,PRD1.BASE,PRD2.SCEEBASE)
// OPTION SYSPARM='00'
// EXEC IESVCSRV,PARM='DD:PRD2.CONFIG(IESVCSRV.Z)'
$$/*
$$/&
$$$$ EOJ
/+
CATALOG  LDVCS.PROC     REPLACE=YES DATA=YES
// EXEC DTRIINIT
   LOAD STARTVCS.Z
/*
/+
/*
// EXEC PROC=LDVCS        TO LOAD  VCS STARTUP  INTO RDR QUEUE
/&
* $$ EOJ
```

Figure 4. Job SKVCSSTJ (for placing Startup Job in Reader Queue)

Job SKVCSCAT – Catalog Members

You can find the job SKVCSCAT in the VSE/ICCF library 59.

You use SKVCSCAT to catalog the configuration members listed in this topic.

```
* $$ JOB JNM=VCSCAT,DISP=D,CLASS=0
// JOB VCSCAT CATALOG VCS CONFIGURATION MEMBERS
// EXEC LIBR,PARM='MSHP'
ACCESS S=PRD2.CONFIG
CATALOG IESVCSRV.Z REPLACE=Y
* $$ SLI ICCF=(SKVCSCFG),LIB=(YY)
/+
CATALOG IESLIBDF.Z REPLACE=Y
* $$ SLI ICCF=(SKVCSLIB),LIB=(YY)
/+
CATALOG IESUSERS.Z REPLACE=Y
* $$ SLI ICCF=(SKVCSUSR),LIB=(YY)
/+
CATALOG IESPLGIN.Z REPLACE=Y
* $$ SLI ICCF=(SKVCSPLG),LIB=(YY)
/+
CATALOG IESSSLCF.Z REPLACE=Y
* $$ SLI ICCF=(SKVCSSSL),LIB=(YY)
/+
/*
/&
* $$ EOJ
```

Figure 5. Job SKVCSCAT (for Cataloging Members for VSE Connector Server)

VSE Library Member SKVCSCFG – General Settings

You can find the skeleton SKVCSCFG in the VSE/ICCF library 59, which you use to specify your general settings for the VSE Connector Server.

Figure 6. Member SKVCSCFG (for Specifying General Settings for VSE Connector Server)

```
; ****************************************************************
; MAIN CONFIGURATION MEMBER FOR VSE CONNECTOR SERVER
; ****************************************************************
; TRACING SPECIFIC SETTINGS:
; - TRACEON : A 32 BIT HEX VALUE PREFIXED WITH '0X'
; 0X00000000 IS OFF, 0XFFFFFFFF IN ON
; - TRACEFILE : DESTINATION FOR TRACE MESSAGES
; DD:SYSLOG, DD:SYSLST OR DD:LIB.SLIB(NAME.TYPE)
; ****************************************************************
TRACEON = 0X00000000 ; TRACE IS OFF
TRACEFILE = DD:SYSLOG ; TRACE GOES TO SYSLOG

; ****************************************************************
; TCP/IP - SERVER SPECIFIC CONFIGURATIONS
; - SERVERPORT : THE TCP PORT WHERE THE SERVER IS LISTENING
; - MAXCLIENTS : THE MAXIMUM NUMBER OF CONCURRENT CLIENTS
; - SSLENABLE : YES/NO - USE SECURE SOCKET LAYER
; ****************************************************************
SERVERPORT = 2893
MAXCLIENTS = 256
SSLENABLE = NO
; SECURITY CONFIGURATION
; - SECURITY: EXTENDED - LOGON, RESOURCE, USER TYPE AND FACILITY
; CHECKING.
; FULL - LOGON, RESOURCE AND USER TYPE CHECKING
; RESOURCE - LOGON AND RESOURCE, BUT NO USER TYPE
; CHECKING.
; LOGON - LOGON, BUT NO RESOURCE AND USER TYPE
; CHECKING
; NO - NO LOGON, RESOURCE AND USER TYPE CHECKING
;
; NOTE: FOR SECURITY = EXTENDED THE FOLLOWING FACILITY RESOURCES
; ARE USED: VSE.CONNECTOR - GENERAL ACCESS
; VSE.CONNECTOR.CONSOLE - CONSOLE ACCESS
; VSE.CONNECTOR.ICCF - ICCF ACCESS
; VSE.CONNECTOR.LIBR - LIBR ACCESS
```

```
; VSE.CONNECTOR.POWER - POWER ACCESS
; VSE.CONNECTOR.VSAM - VSAM ACCESS
; RESOURCE VSE.CONNECTOR IS CHECKED FOR READ AUTHORITY
; ONLY. ALL OTHER FACILITIES ARE CHECKED ACCORDING TO
; THE DESIRED ACCESS REQUEST (READ, UPDATE, ALTER)
; *****************************************************************
SECURITY = FULL

; *****************************************************************
; TIMEOUT FOR VSAM AUTO CLOSE
; - AUTOCLOSE : THE TIMEOUT IN SECONDS FOR VSAM AUTOCLOSE
; *****************************************************************

; *****************************************************************
; DEFAULT CLASS FOR JOBS
; - DEFAULTCLASS : THE JOB CLASS WHERE UTILITY JOBS SHOULD RUN
; *****************************************************************
DEFAULTCLASS = P
; *****************************************************************
; CODE PAGE CONVERSIONS
; - ASCII_CP : ASCII CODE PAGE
; - EBCDIC_CP : EBCDIC CODE PAGE
; *****************************************************************
ASCII_CP = IBM-850
EBCDIC_CP = IBM-1047

; *****************************************************************
; SYSTEM LANGUAGE
; - LANGUAGE : THE SYSTEM'S LANGUAGE (E, J)
; *****************************************************************
LANGUAGE = E

; *****************************************************************
; DESCRIPTION SENT AS IDENTIFY
; - DESCRIPTION : THIS STRING IS SENT AS IDENTIFY TO THE CLIENT
; CHANGE '<YOUR SYSTEM>' TO YOUR SYSTEM'S NAME
; *****************************************************************
DESCRIPTION = VSE CONNECTOR SERVER ON <YOUR SYSTEM>

; *****************************************************************
; MESSAGES CONFIGURATION
; THIS SETTING CONTROL WHERE THE CONNECTION AND USER MESSAGES ARE
; WRITTEN TO.
; - SECURITY: OFF - NO MESSAGES WILL BE SHOWN.
; SYSLOG - THE MESSAGES WILL GO TO SYSLOG
; SYSLST - THE MESSAGES WILL GO TO SYSLST
; BOTH - THE MESSAGES WILL GO TO SYSLOG AND SYSLST
; *****************************************************************
MESSAGES = SYSLOG

; *****************************************************************
; SUB CONFIGURATION MEMBERS NEEDED FOR VSE CONNECTOR SERVER
; - LIBRCFGFILE : LIBRARY DEFINITION FILE. CONTAINS THE LIBRARIES
; THAT ARE VISIBLE FOR THE VSE CONNECTOR SERVER.
; - USERSCFGFILE : USER/SECURITY CONFIG FILE. DEFINES ADDITIONAL
; SECURITY FOR USERS AND IP ADDRESSES.
; - PLUGINCFGFILE : PLUGIN CONFIG FILE. DEFINES THE PLUGINS THAT
; ARE LOADED AT SERVER STARTUP.
; - SSLCFGFILE : SSL CONFIG FILE. DEFINES SSL PARAMETERS
; NOTE: YOU HAVE TO CHANGE THE NAMES AND LOCATIONS OF THESE MEMBERS
; IN THIS MEMBER IF YOU MOVE THEM TO ANOTHER LIBRARY!
; *****************************************************************
LIBRCFGFILE = DD:PRD2.CONFIG(IESLIBDF.Z)
USERSCFGFILE = DD:PRD2.CONFIG(IESUSERS.Z)
PLUGINCFGFILE = DD:PRD2.CONFIG(IESPLGIN.Z)
SSLCFGFILE = DD:PRD2.CONFIG(IESSSLCF.Z)
```

VSE Library Member SKVCSLIB – Specify Libraries to Be Accessed

You can find the skeleton SKVCSLIB in the VSE/ICCF library 59.

You use SKVCSLIB to specify the libraries that can be accessed by the VSE Connector Server. This skeleton consists of a list of libraries, which you can extend or restrict according to your own requirements. You must, however, enter each library name on a separate line.

```
PRD1
PRD2
PRIMARY
IJSYSRS
```

Figure 7. Member SKVCSLIB (for Specifying Libraries to Be Accessed by VSE Connector Server)

VSE Library Member SKVCSPLG – Specify Plugins to Be Loaded

You can find the skeleton SKVCSPLG in the VSE/ICCF library 59.

You use SKVCSPLG to specify the VSE Connector Server plugins to be loaded, when the VSE Connector Server is started.

This is the syntax of the PLUGIN statement:

```
►►──PLUGIN=phase_name──,──PARM=──┬──────────────────┬──────────────────────►◄
                                 └─parameter_string─┘
```

Figure 8 shows the member you use to specify VSE Connector Server plugins:

```
* *******************************************************************
* VSE CONNECTOR SERVER PLUGIN CONFIGURATION MEMBER
* *******************************************************************
* THE FOLLOWING PLUGINS ARE LOADED DURING STARTUP OF THE SERVER
* UNCOMMENT THE SAMPLES BELOW TO CHANGE THEM
* *******************************************************************
PLUGIN=IESSAPLG,PARM=CICS=F2,CONS=IESA,TRANS=IEXM,EXIT=IESSAEXT
PLUGIN=IESHTOHP,PARM=
PLUGIN=IESCOMPH,PARM=
PLUGIN=IESVSAPL,PARM=
PLUGIN=IESDLIPL,PARM=
* PLUGIN=SAMPLE,PARM=MY PARAMTER STRING
* PLUGIN=<PHASE NAME>,PARM=<PARAMETER STRING>
```

Figure 8. Member SKVCSPLG (for Specifying Plugins for VSE Connector Server)

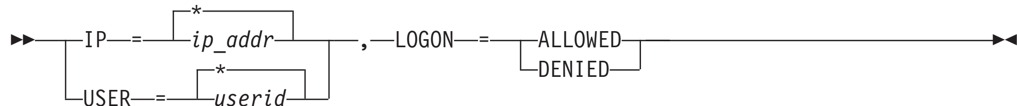
For details of how to write your own plugins, see Chapter 21, “Extending the Java-Based Connector,” on page 269.

VSE Library Member SKVCSUSR – Specify Logon Access

You can find the skeleton SKVCSUSR in the VSE/ICCF library 59.

You use SKVCSUSR to specify the users, or groups of users, who can logon to the VSE Connector Server. Using this skeleton, you can also prevent specific users or IP addresses from being able to access the VSE Connector Server.

This is the syntax of this member:




```
* ****************************************************************
* VSE CONNECTOR SERVER USER SECURITY CONFIGURATION MEMBER
* YOU CAN EITHER ALLOW OR DENY THE LOGON FOR A SPECIFIED
* USER OR IP OR GROUP OF USERS AND IP ADDRESSES.
* ****************************************************************
* USERS FROM THIS IP'S ARE ALLOWED TO LOGON
* UNCOMMENT THE SAMPLES AND MODIFY THEM
* ****************************************************************
IP   = *,            LOGON = ALLOWED
* IP = 9.164.123.456, LOGON = DENIED
* IP = 9.165.*      , LOGON = DENIED
* IP = 10.0.0.*     , LOGON = ALLOWED
* ****************************************************************
* THIS USERS ARE ALLOWED TO LOGON
* UNCOMMENT THE SAMPLES AND MODIFY THEM
* ****************************************************************
USER = *,            LOGON = ALLOWED
* USER = BOBY,       LOGON = ALLOWED
* USER = SYS*,       LOGON = DENIED
```

Figure 9. Member SKVCSUSR (for Specifying Logon Access to VSE Connector Server)

Notes:

1. If you do not make any entries in Figure 9, *no* access authorizations will be defined!
2. You can use the wildcard (*) within an IP address or user name.

VSE Library Member SKVCSSSL – Configure for SSL

You can find the skeleton SKVCSSSL in the VSE/ICCF library 59.

You use SKVCSSSL to specify the:

- Version of SSL to be used.
- Name of the *VSE Keyring Library* to be used by the VSE Connector Server (for details, refer to the chapter "Preparing Your System to Use SSL" in the *z/VSE Administration*).
- Name of the *server certificate* to be used by the VSE Connector Server (for details, see chapter "Preparing Your System to Use SSL" in the *z/VSE Administration*).
- *Session timeout* in seconds – the number of seconds that the VSE Connector Server allows a VSE Connector Client to reconnect, without requiring a full *handshake*.

```
; ******************************************************************
;      SSL CONFIGURATION MEMBER FOR VSE CONNECTOR SERVER
; ******************************************************************

; ******************************************************************
; SSLVERSION   SPECIFIES THE MINIMUM VERSION THAT IS TO BE USED
;              POSSIBLE VALUES ARE:   SSL30 AND TLS31
; KEYRING      SPECIFIES THE SUBLIBRARY WHERE THE KEY FILES ARE
;              STORED.
; CERTNAME     NAME OF THE CERTIFICATE THAT IS USED BY THE SERVER
; SESSIONTIMEOUT NUMBER OF SECONDS THAT THE SERVER WILL USE TO
;              ALLOW A CLIENT TO RECONNECT WITHOUT PERFORMING A
;              FULL HANDSHAKE. (86440 SEC = 24 HOURS)
; AUTHENTICATION TYPE OF AUTHENTICATION. POSSIBLE VALUES ARE:
;              SERVER - SERVER AUTHENTICATION ONLY
;              CLIENT - SERVER AND CLIENT AUTHENTICATION
; ******************************************************************
SSLVERSION     = SSL30
KEYRING        = CRYPTO.KEYRING
CERTNAME       = SAMPLE
SESSIONTIMEOUT = 86440
AUTHENTICATION = SERVER

; *****************************************************************
; CIPHERSUITES SPECIFIES A LIST OF CIPHER SUITES THAT ARE ALLOWED
; *****************************************************************
CIPHERSUITES =   ; COMMA SEPARATED LIST OF NUMERIC VALUES
               01,  ; RSA512_NULL_MD5
               02,  ; RSA512_NULL_SHA
               08,  ; RSA512_DES40CBC_SHA
               09,  ; RSA1024_DESCBC_SHA
               0A,  ; RSA1024_3DESCBC_SHA
               62,  ; RSA1024_EXPORT_DESCBC_SHA
               2F   ; TLS_RSA_WITH_AES_128_CBC_SHA
               35   ; TLS_RSA_WITH_AES_256_CBC_SHA
```

Figure 10. Member SKVCSSSL (for Configuring the VSE Connector Server for SSL)

Configuring the Date-Format for the VSE Connector Server

The VSE Connector Server obtains the current date-format from the *LE/VSE run-time options*. The default is **mm/dd/yyyy** (the USA date-format). If you have changed the date-format of your z/VSE system, you *must* also change the LE/VSE date-format. Otherwise, VSE Connector Client applications will obtain incorrect values for the day, month, and year!

To change the LE/VSE date-format, you should:

1. Change this parameter in the member CEEDOPT (which is stored in VSE/ICCF library 62):
   ```
   COUNTRY=((US),OVR),
   ```
2. Catalog the new value by submitting the job template CEEWDOPT,(which is stored in VSE/ICCF library 62).

Alternatively, you can specify online the date-format for the VSE Connector Server, as shown in this example:

```
// EXEC IESVCSRV,PARM='COUNTRY(DE)/DD:PRD2.CONFIG(IESVCSRV.Z)'
```

For details of:

- how to specify LE/VSE run-time options,
- the country codes that you can use,

refer to the *LE/VSE Customization Guide*, SC33-6682.

Starting the VSE Connector Server

You start the VSE Connector Server by releasing the startup job STARTVCS from the VSE/POWER reader queue. By default, the server then runs in a partition of dynamic class R.

STARTVCS is placed in the POWER reader queue either:

- During initial installation of z/VSE
- During a cold startup of z/VSE

Note: TCP/IP Must Be Running! This job waits for TCP/IP to start up. Refer to *z/VSE TCP/IP Support* for details of how to start TCP/IP.

When started, the VSE Connector Server listens to incoming TCP/IP traffic on port **2893**:

- The server's security subsystem uses RACROUTE requests to check logon and resource authorization.
- The server can handle multiple concurrent clients.

Figure 11 shows the startup job STARTVCS:

```
* $$ JOB JNM=STARTVCS,CLASS=R,DISP=L
* $$ LST CLASS=A,DISP=D
// JOB STARTVCS START UP VSE CONNECTOR SERVER
// ID USER=VCSRV
*   WAITING FOR TCP/IP TO COME UP
// EXEC REXX=IESWAITR,PARM='TCPIP00'
// LIBDEF *,SEARCH=(PRD2.CONFIG,PRD1.BASE,PRD2.SCEEBASE)
// OPTION SYSPARM='00'
// EXEC IESVCSRV,PARM='DD:PRD2.CONFIG(IESVCSRV.Z)'
/*
/&
* $$ EOJ
```

Figure 11. Startup Job STARTVCS (for Starting the VSE Connector Server)

Testing the Communication Between VSE Connector Client and Connector Server

Once you have completed the installation of the VSE Connector Client and VSE Connector Server, you can test if these two parts of the Java-based connector communicate correctly with each other by running some pre-compiled sample Java applications.

Before you begin, you must ensure that:

- TCP/IP for VSE/ESA is running on your z/VSE host.
- The VSE Connector Server is running on your z/VSE host.
- A TCP/IP connection exists between your local workstation and the z/VSE host.

The VSE Connector Client installation includes a set of batch files that run on Windows and Unix/Linux operating systems. They are stored in **samples**, which is a sub-directory below the directory where you installed the VSE Connector Client.

Each of the batch files contained in **samples** runs one sample *Java application* or *sample applet*. To run a sample, you should:

1. Open a command prompt.
2. Proceed to the **samples** sub-directory.
3. Run any batch file (you are not required to enter any parameters).

To obtain a list (with descriptions) of all the samples you can run, you should:

1. Start the VSE Connector Client.
2. Click **Run examples** in the left frame of this window.

Obtaining a List of VSE Connector Server Commands

To obtain a list of the VSE Connector Server commands that you can use, simply enter **Help** or **?** at the operator console. Figure 12 shows what will be displayed.

```
SYSTEM:  z/VSE               z/VSE 4.3       TURBO (01)      USER:  JSCH
                                                             TIME:  14:25:53
F7-0112 IPN300I Enter TCP/IP Command
msg startvcs,data=?
AR 0015 1I40I  READY
R1 0045 IESC1043I HELP COMMAND
R1 0045   HELP|?                                 PRINTS THIS MESSAGE
R1 0045   STATUS [ALL|CONFIG|CLIENTS|PLUGINS|VSAM] PRINTS STATUS INFORMATION
R1 0045   SENDMSG <USER(S)> <MESSAGE TEXT>        SENDS A MESSAGE TO A USER
R1 0045   SHUTDOWN [NOPROMPT]                     SHUTS DOWN THE SERVER
R1 0045   SETTRACE <TRACEFILE> <TRACELEVEL>       SET TRACING ON/OFF
R1 0045   STOP CLIENT <CLIENT-ID|ALL>             STOPS THE SPECIFIED CLIENT
R1 0045   CLOSE VSAM <SLOT-ID|ALL>                CLOSES A VSAM CLUSTER


==>

1=HLP 2=CPY 3=END 4=RTN 5=DEL 6=DELS 7=RED 8=CONT 9=EXPL 10=HLD 11=PCUU 12=RTRV
```

Figure 12. Displaying the Commands Provided by the VSE Connector Server

Entering a Command for the VSE Connector Server

To enter a command that the VSE Connector Server should process, you should use this command syntax:

```
msg <jobname>,data=<command>
```

where:

- *<jobname>* is the actual name of the VSE Connector Server startup job.
- *<command>* is one of the command strings shown in Figure 12.

For example:

```
msg startvcs,data=status
```

Maintaining Security Using the VSE Connector Server

When a Web application uses the VSE Java Beans class library (described in "Example of Using VSE Java Beans to Connect to the Host" on page 148) to connect to the z/VSE host, the Web application must first perform a logon to the z/VSE host (supplying a valid user ID and password). Providing this logon is successful, the Web application obtains its access rights from the supplied user ID.

When the VSE Connector Server receives a request from a Web application, it passes this request to the currently-active VSE security manager (either the Basic Security Manager or External Security Manager). The security manager then checks whether or not the Web application should be allowed to access the requested resources or data.

The VSE Connector Server also uses a configuration file SKVCSUSR, which contains two lists of user IDs and IP addresses that:

1. *are* allowed to connect to the VSE Connector Server.
2. *are not* allowed to connect to the VSE Connector Server.

This is explained in "VSE Library Member SKVCSUSR – Specify Logon Access" on page 32.

When you write *servlets*, you might use a special z/VSE user ID which then allows the servlet to connect to the VSE Connector Server, without forcing the end-user to logon to the VSE Connector Server. Using this user ID, your servlets can restrict the type of requests, and also restrict access to data.

When you write *applets*, you should ***never*** "hard-code" any user IDs and passwords in the applet code: when the applet is downloaded to a Web browser and is stored in the Web browser's cache, this information could possibly be displayed by unauthorized persons (hackers).

TCP/IP for VSE/ESA also contains its own security functions. However, these functions are applicable only when using TCP/IP applications such as FTP, HTTP, or Telnet daemons. The VSE Connector Server *always* communicates with the VSE security manager.

Note: Security between the physical/logical middle-tier and Web clients is established using SSL. SSL is, however, *not* used between the physical/logical middle-tier and your z/VSE host.

Chapter 6. Configuring DL/I for Access Via VSE Java Beans

This chapter describes how you configure your DL/I system so that it can be accessed via VSE Java Beans.

It contains these main topics:
- "Host Installation Activities That Must Be Already Completed"
- "Step 1: Skeleton SKDLISMP – Define Sample Database"
- "Step 2: Customize CICS TS" on page 38

Three VSE Java Beans are used for DL/I access: VSEDli, VSEDliPsb, and VSEDliPcb. For descriptions of these VSE Java Beans, see "Contents of the VSE Java Beans Class Library" on page 143.

For an example of the coding you can use to access DL/I data via VSE Java Beans, see "Example of Using VSE Java Beans to Access DL/I Data" on page 159.

Host Installation Activities That Must Be Already Completed

This topic provides a summary of the installation activities that must already be completed on the z/VSE host, before you use VSE Java Beans to access DL/I data.
- The AIBTDLI interface must be installed for accessing DL/I data via VSE Java Beans. To use the AIBTDLI interface:
  - DL/I VSE must be installed.
  - Your CICS/DLI system must have all databases (DBDs) that you wish to use defined to CICS, together with the AIBTDLI interface.
  - Your CICS/DLI system must have:
    - all PSBs defined in the DL/I online nucleus DLZNUCxx
    - an active MPS system.
  - The DL/I task termination exit *DLZBSEOT* (described in "Task Termination and Abend Handling" on page 410) must be resident in the SVA.

Step 1: Skeleton SKDLISMP – Define Sample Database

Use skeleton SKDLISMP (available in VSE/ICCF library 59) to define and load a DL/I sample database, if you:
- wish to use the IBM-supplied sample database for testing and learning purposes
- have not already defined and loaded this database during previous installations.

The skeleton SKDLISMP contains these jobs:
1. STJDBDGN (generate the DBDs for the sample database).
2. STJPSBGN (generate the PSBs for the sample database).
3. STJACBGN (generate the ACBs for the sample database).
4. STJPREOR (a pre-reorganization utility).
5. STJDFINV (define the cluster for the sample database).
6. STJLDCST (load the sample database).

© Copyright IBM Corp. 2000, 2013

7. STJPRRES (prefix resolution).
8. STJPRUPD (prefix update).

Step 2: Customize CICS TS

To access DL/I data via VSE Java Beans, you must customize a CICS TS - DL/I online system (providing you have not already done so during previous installation activities):

1. Configure your CICS/DLI online system, as described in:
   - Part 6 of the *DL/I Resource Definition and Utilities* publication.
   - The topic "CICS – DL/I Tables – Requirements" of the *DL/I Resource Definition and Utilities* publication.
   - Topic "Migrating to DL/I VSE 1.11 and the CICS Transaction Server for VSE/ESA 1.1" of the *DL/I Release Guide*.
2. Define in the CICS FCT the sample database STDIDBP and other databases you wish to access.
3. Provide labels for the sample database STDIDBP (// DLBL STDIDBC ...) and other databases you wish to access.
4. Create a new DL/I online nucleus (DLZACT generation), by including all DL/I online programs and PSBs that you wish to use. The CICS/DLI mirror program DLZBPC00 must be authorized for PSB STBICLG, used for accessing the DL/I sample database. The CICS/DLI mirror program DLZBPC00 must also be authorized for any other PSBs you wish to use for accessing other DL/I databases.
5. Account for an increased number of concurrent DLZBPC00 mirror tasks in the CICS/DLI online system: you must accordingly adjust the MAXTASK and CMAXTSK parameters in the DLZACT generation.
6. Load the DL/I exit routine DLZBSEOT into the SVA.
7. Start an MPS system.

DLZMPX00 is SVA-eligible, and is used for accessing DL/I data via the *AIBTDLI interface* (see "Overview of the AIBTDLI Interface" on page 403 for an explanation of DLZMPX00 and the AIBTDLI interface). The AIBTDLI interface uses DLZMPX00 from the SVA (if it resides there), or loads DLZMPX00 into partition space and uses it from there.

Chapter 7. Installing the VSE Script Connector

This chapter first provides an overview of how the VSE Script Connector is used. Then it describes how you install the *server-part* (the VSE Script Server) of the VSE Script Connector.

The *client-part* of the VSE Script Connector (the VSE Script Client) is *not* installed here, but can be either:
- a user-written Java application.
- a user-written non-Java application.
- an existing office product, such as a word-processing or spreadsheet program.

This chapter contains these main topics:
- "Overview of the VSE Script Connector"
- "Step 1: Download the Install-File and Perform the Installation" on page 40
- "Step 2: Configure the VSEScriptServer Properties File" on page 42
- "Step 3: Configure the Connections Properties File" on page 43

Notes:
1. The VSE Script Server must run on a Java-enabled platform.
2. The client-part (the VSE Script Client) can run on either a Java-enabled platform **or** a non-Java-enabled platform.
3. You are **not** required to write your own Java code in order to use the VSE Script Connector. Instead, you simply write your own VSE Scripts, using the IBM-supplied VSE Script Language.

Related Topic:
- Chapter 25, "Using the VSE Script Connector for Non-Java Access," on page 437

Overview of the VSE Script Connector

As described in "Overview of the Java-Based Connector" on page 23, VSE Java Beans provide direct access to the z/VSE host from any kind of Java program (servlets, applets, EJBs, and so on) running on a *Java platform*. In addition, you can use the *VSE Script connector* to access z/VSE host data from *non-Java* platforms. This is the main advantage of using the VSE Script connector (although it can also be used to access z/VSE host data from most Java platforms).

The VSE Script connector is supplied as *part of* the Java-based connector. It can only be used in a 3-tier environment (explained in "Overview of 3-Tier Environments" on page 8), and consists of:
- a VSE Script Client running on a Java or non-Java platform, and which can be either:
  - a user-written Java application (for example a Web-service).
  - a user-written non-Java application (for example a Windows C-program, a Windows CGI-program, or a COBOL application).
  - an office product, such as a word-processing or spreadsheet program (for example Lotus 1-2-3 or Lotus WordPro), where for example, a Visual Basic script is used to call a VSE Script.
- the VSE Script Server running on the physical/logical middle-tier of a 3-tier environment, which interprets and executes VSE Script files.

© Copyright IBM Corp. 2000, 2013

- online documentation, including a programming reference manual.

The VSE Script connector works in this general way:

1. The VSE Script Client calls a VSE Script, to make a request for data stored on the z/VSE host. These VSE Script (batch) files contain statements written using the VSE Script language, which is a special programming language. The VSE Script language can be used in any environment (even in Visual Basic scripts).
2. The VSE Script Server running on a Java-enabled physical/logical middle-tier platform then reads, interprets, and translates, the VSE Script file statements into VSE Java Beans requests. The VSE Script Server uses the VSE Java Beans to connect to the VSE Connector Server running on the z/VSE host, and to forward the VSE Java Beans requests.
3. The VSE Connector Server accesses the required z/VSE data and functions, and sends the reply back to the VSE Script Server.
4. The VSE Script Server converts the data to the format that the VSE Script Client can use, and returns the data to the VSE Script Client.

Step 1: Download the Install-File and Perform the Installation

You install the VSE Script Server on a Java-enabled platform.

Note: Before you begin, you must already have installed the Java Development Kit (JDK) 1.5 or later on the development platform where you plan to install the VSE Script Server. If you do not have JDK 1.5 or later installed, refer to "Installing and Configuring Java" on page 19 for details of how to install it.

Step 1.1: Obtain a Copy of the VSE Script Server

To obtain a copy of the VSE Script Server, you must decide if you wish to obtain it:
- From the Internet.
- By installing the VSE Connectors Workstation Code component from the Extended Base Tape.

To obtain the VSE Script Server *from the Internet*, you should:

1. Start your Web browser and go to URL:

   `http://www.ibm.com/systems/z/os/zvse/downloads/`
2. From within the VSE Script Server section, download the file **vsescript*nnn*.zip** to the directory where you wish to install the VSE Script Server. **Note:** *nnn* refers to the current VSE version (for example, **vsescript430.zip**).

To obtain the VSE Script Server *by installing the VSE Connectors Workstation Code component*, you should:

1. Install the VSE Connectors Workstation Code component from the Extended Base Tape. After you have installed this component, the VSE Script Server W-book **iesscrpt.w** will be stored in z/VSE sublibrary PRD2.PROD.
2. Use the FTP (file transfer program) utility of TCP/IP for VSE/ESA to download **iesscrpt.w** to the directory where you wish to install the VSE Script Server.

Notes:

1. You must download **iesscrpt.w** in *binary*.
2. Make sure that Unix mode is ***turned off***. Otherwise **iesscrpt.w** will be downloaded in ASCII mode, even when you specify *binary*. *Unix mode* is one parameter of your VSE FTP daemon. Some FTP clients might *force* Unix mode

to be turned on! The example below shows how a successful transfer of **iesscrpt.w** was made using a batch FTP client. The place where the UNIX mode is set, is shown as bold.

```
c:\temp>ftp 9.164.155.2
Connected to 9.164.155.2.
220-TCP/IP for VSE -- Version 01.05.F  -- FTP Daemon
    Copyright (c) 1995,2006 Connectivity Systems Incorporated
220 Service ready for new user.
User (9.164.155.2:(none)): sysa
331 User name okay, need password.
Password:
230 User logged in, proceed.
ftp> cd prd2
250 Requested file action okay, completed.
ftp> cd prod
250 Requested file action okay, completed.
ftp> binary
200 Command okay.
ftp> get iesscrpt.w
200 Command okay.
150-File: PRD2.PROD.IESSCRPT.W
    Type: Binary Recfm: FB Lrecl:    80 Blksize:      80
    CC=ON  UNIX=OFF RECLF=OFF TRCC=OFF CRLF=ON  NAT=NO
150 File status okay; about to open data connection
226-Bytes sent:       2,378,200
    Records sent:        29,728
    Transfer Seconds:         8.26 (    290K/Sec)
    File I/O Seconds:         1.97 (  1,548K/Sec)
226 Closing data connection.
2378200 bytes received in  8,56 seconds (277,82 Kbytes/sec)
ftp> bye
221 Service closing control connection.
c:\temp>ren iesscrpt.w vsescript.zip
```

Step 1.2: Perform the Installation of the VSE Script Server

To perform the installation of the VSE Script Server:

1. Unzip the file **vsescript.zip**, which contains these files:
   - setup.jar (contains the VSE Script Connector code)
   - setup.bat (an install batch file for Windows)
   - setup.cmd (an install batch file for OS/2)
   - setup.sh (an install script for Linux/Unix)
2. Start the batch file (by double-clicking the file) that is applicable to your operating-system platform.
3. The installation process now begins, and you are guided through various installation menus.
4. To access the HTML-based documentation, you can now use your Web browser to open the file **server.html** in the **/doc** subdirectory.
5. To use SSL, you must have:
   - Enabled SSL in TCP/IP for VSE/ESA.
   - Created the required keys and certificates.

   Note: The VSE Script Server supports SSL-encrypted connections *from z/VSE 4.2 onwards*. This support is for connections from VSE Script Clients to the VSE Script Server, and from the VSE Script Server to z/VSE.

Step 2: Configure the VSEScriptServer Properties File

The properties file for the VSE Script Server is called **VSEScriptServer.properties**, which is a text file that you can edit using any text editor.

Comment lines begin with a # character in the first column.

These are the settings that you define in **VSEScriptServer.properties**:

messages= on|off
: If you define **messages= on**, all messages will be printed. If you define **messages= off**, messages will not printed (and "quiet mode" will be active).

listenport = *TCP/IP portnumber*
: Port number which the VSE Script Server uses to listen for requests.

maxconnections = *number*
: Maximum number of simultaneous connections that are allowed from VSE Script Clients.

scriptdirectory = ./scripts
: Root directory to contain the Scripts.

connectionconfig = Connections.properties
: Name of the connection configuration file (described in "Step 3: Configure the Connections Properties File" on page 43).

codepage
: Code page for VSE Script Clients. If not specified; the current default code page is used:

sslversion[1]
: Can be either SSL or TLS.

clientauthentication[1]
: If client authentication is `true`, the VSE Script Server requests the VSE Script Client to send its certificate, and verifies it. If client authentication is `false` or not specified, client authentication is not performed.

keyringfile[1]
: File name of the keyring file.

keyringpwd[1]
: Password that is used to open the keyring file.

ciphersuites[1]
: List of cipher suites (separated by commas) that are accepted by the VSE Script Server. If this property is not specified, *all* supported cipher suites can be used.

If you wish to use a different properties file than the default, you must specify the file name of your properties file as a parameter, using this command.

```
java com.ibm.vse.script.VSEScriptServer MyPropertiesFile.properties
```

1. For use with SSL-encrypted communication between VSE Script Client and VSE Script Server. To use SSL, you must have created the required keys and certificates.

Step 3: Configure the Connections Properties File

You define each connection from the VSE Script Server to the z/VSE host using a set of *five* properties (`connection.1.name`, up to `connection.1.password`) contained in a properties file. This is a text file that you can edit using any text editor. For an overview of how these connections are used, see Figure 176 on page 438.

Here is an example of a Connections properties file:

```
#Connection.properties

connection.1.name=vsecon
connection.1.ip=9.164.155.2
connection.1.port=2893
connection.1.userid=fran
connection.1.password=mypasswd

connection.2.name=vsefran
connection.2.ip=9.164.155.95
connection.2.port=2893
connection.2.userid=sysa
connection.2.password=mypasswd

 ...
connection.timeout=100
connection.2.sslversion=SSL
connection.2.ciphersuites=TLS_RSA_WITH_AES_128_CBC_SHA,
SSL_RSA_WITH_3DES_EDE_CBC_SHA,SSL_RSA_WITH_DES_CBC_SHA,
SSL_RSA_WITH_NULL_SHA,SSL_RSA_WITH_NULL_MD5,
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
connection.2.keyringfile=keyring.pfx
connection.2.keyringpwd=ssltest
connection.2.logonwithcert=no
 ...
connection.timeout=100
 ...
```

Comment lines begin with a # character in the first column.

These are the settings that you define in **Connections.properties** for *each* connection.

name = *name*
: Logical name you give to the z/VSE host. Your VSE Scripts will refer to this name when accessing the z/VSE host. For an example of how **name** is used, see “Step 4: Modify the Sample VSE Script” on page 447.

ip = *IP address*
: TCP/IP address of the z/VSE host to which the VSE Script Server is to connect.

port = *number*
: Port number used by VSE Connector Server to listen for incoming requests.

userid = *user-ID*
: z/VSE user-ID used by the VSE Script Server to build a connection to the z/VSE host.

password = *password*
: The password you wish to assign to the connection from the VSE Script Server to the z/VSE host. During the initial startup of the VSE Script Server, the password is encrypted and stored using the property **connection.n.encpassword**.

sslversion[2] **= SSL30 | TLS31**
: If specified, you must also specify the other SSL-specific properties (below).

ciphersuites[2] **=** *list of cipher suites* **(separated by commas)**
: These are the cipher suites that are currently supported by z/VSE:
TLS_RSA_WITH_AES_128_CBC_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA
SSL_RSA_WITH_DES_CBC_SHA
SSL_RSA_WITH_NULL_SHA
SSL_RSA_WITH_NULL_MD5
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA

keyringfile[2] **=** *name*
: File name of the keyring file that contains the keys and certificates.

keyringpwd[2] **=** *password*
: The password used to encrypt the keyring file. During the initial startup of the VSE Script Server, the password is encrypted and stored using the property `connection.n.enckeyringpwd`.

logonwithcert[2] **= true | false**
: If set to `true`, the VSE Script Client's certificate is used to sign-on to z/VSE. This requires that client authentication is enabled on z/VSE.

In addition, you can define in **Connections.properties** this global setting:

connection.timeout = *seconds*
: The time in seconds before an unused connection in the pool (to the z/VSE host) is closed and destroyed.

2. For use with SSL-encrypted communication between the VSE Script Server and the z/VSE system. To use SSL, you must have created the required keys and certificates.

Chapter 8. Installing the VSAM Redirector Connector

This chapter describes how you install and implement the VSAM Redirector connector.

It contains these main topics:

- “Overview of the VSAM Redirector Connector”
- “VSAM Integration Considerations” on page 49
- “Configuring the VSAM Redirector Client / VSAM Capture Exit” on page 49
- “Installing the VSAM Redirector Server” on page 63
- “Using the IBM-Provided VSAM Redirector Handlers” on page 67
- “Using the IBM-Provided VSAM Redirector Loaders” on page 69
- “IBM-Supplied Example of DB2-Related Handler” on page 70

Overview of the VSAM Redirector Connector

The VSAM Redirector Connector enables VSE programs to work with:
- VSAM data that is synchronized with a remote database or file system.
- VSAM files, all of whose data has been moved to a remote platform.

Using the VSAM Redirector Connector:
- VSAM data can be migrated to other file systems or databases.
- Data can be synchronized on different systems with VSE VSAM data.
- VSE programs can work transparently with data on other file systems or databases.
- Changes made to VSAM data can be captured and temporarily stored on your z/VSE system for further processing.

Your existing z/VSE host programs that are:
- written in any language (COBOL, PL/I, ASSEMBLER)
- batch or CICS programs

can work with migrated VSAM data without the need to amend and recompile these z/VSE host programs. The VSAM Redirector Connector manages all connections and data conversions.

The VSAM Redirector Connector consists of:
- The *VSAM Redirector Client* for *synchronous* redirection, which is installed on your *z/VSE host*. It is responsible for communication and redirection of VSAM requests.
- The *VSAM Capture Exit* for *asynchronous* redirection, which is installed on your *z/VSE host*. It captures changes made to a specific VSAM file, and then stores them in:
  - Another VSAM file (the VSAM delta cluster).
  - A WebSphere MQ queue.
- The *VSAM Redirector Server* which is installed on each remote *Java platform*, as shown in Figure 13 on page 47. It is a Java program that:
  1. Is responsible for connection-handling.
  2. Is responsible for data conversion.

© Copyright IBM Corp. 2000, 2013

3. Builds the interface to the different file system *VSAM Redirector Handlers* (referred to simply as *redirector handlers*.
4. Generates error messages that are the same as those generated before VSAM datasets were migrated. Therefore, your application programs do not need to be amended for changes in error-message handling.

- *VSAM Redirector Loaders* (referred to simply as *redirector loaders*), which are Java programs on the remote platform which are used for loading and processing VSAM data. The following redirector loaders are supplied by IBM:
  – A *basic redirector loader*, which is used to load data from a given VSAM file and process it via a specified redirector handler (for example, to store the data in a database).
  – A *delta redirector loader*, which is used to load captured data from a VSAM delta cluster and process it via a specified redirector handler.
  – An *MQ redirector loader*, which is called from WebSphere MQ on a workstation as an “MQ trigger”, to process captured data via a specified redirector handler.
- A *graphical user interface*, which you can use to configure the data mapping used for the `DBHandler` redirector handler either manually, or by importing COBOL copybooks. You can also use the graphical user interface to create the corresponding database tables.

Redirector handlers are also stored on the *Java platform*, and have a common interface. They are specific to the file system with which they work. For all connections, information about the file and the request are sent to the redirector handler.

Using *synchronous data redirection*, you can migrate or synchronize your VSAM data for example with DB2 tables residing on a remote system, and your VSE programs will then work with this data, without requiring any changes to these VSE programs. On the remote system, a *Java handler* provides access to the specific file system or database on the remote system. For example, you can migrate your VSAM data into DB2 tables residing on a remote system, and your VSE programs will then work with this data, without requiring any changes to these VSE programs.

Figure 13 on page 47 provides an overview of how the VSAM Redirector Connector can be used to *synchronously* redirect VSAM requests from an application running on the z/VSE host to a DB2 database stored on a Java platform. It can use any of these IBM-supplied redirector handlers on the Java platform:
- DB2Handler
- DBHandler
- CSVFileHandler

The characteristics of each handler are explained in “Using the IBM-Provided VSAM Redirector Handlers” on page 67.

Each handler can decide at each of the above requests, which processing is required for the remote data.

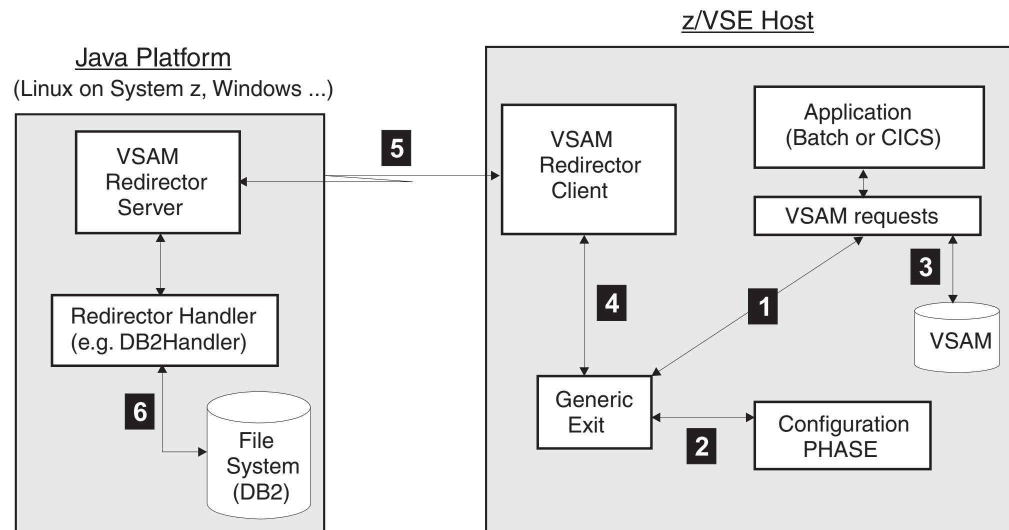


Figure 13. How Synchronous Data Redirection Is Used

The general processing shown in Figure 13 is as follows:

1 An application running on the z/VSE host issues a VSAM command (for example, to open a VSAM file). The request is passed to the *generic exit* (IKQVEX01.PHASE).

2 The generic exit checks whether the VSAM file has been set up to be redirected. To do so, it checks the configuration phase (IESRDCFG.PHASE).

3 If the VSAM file has not been redirected to another Java platform (and is therefore still stored as a VSAM record on the z/VSE host), the generic exit returns and indicates that the VSAM file has not been redirected. It also indicates that the generic exit should not be called again for any request against *this* VSAM file. Normal VSAM processing then continues.

4 If the VSAM file *has* been redirected to another Java platform, the generic exit (IESREDIR.PHASE) calls the VSAM Redirector Client.

5 The VSAM Redirector Client establishes a connection to the VSAM Redirector Server running on the Java platform, and forwards the VSAM file request, together with any data (such as a VSAM record contents) to the VSAM Redirector Server.

6 The VSAM Redirector Server uses the redirector handler that is specified in the configuration phase, to perform access to the target file system or database. In Figure 13, the specified redirector handler is *DB2Handler* (which is supplied by IBM during the installation of the VSAM Redirector Server). *DB2Handler* implements access to a DB2 database.

Asynchronous redirection of VSAM data is provided by the *VSAM Capture Exit*, which allows you to:
1. Capture changes made to a particular VSAM file.
2. Store these changes for further processing.

There are two options you can use:
- Save changes in a VSAM cluster (the VSAM delta cluster).
- Save changes in an WebSphere MQ queue.

Captured changes are written as "delta records" or messages. They contain:
- The data of the changed record.
- Information about *when* (the timestamp) and *by whom* (the partition, phase name, origin value, and so on) the record was changed.

The saved changes can later be accessed via the Java-based connector by Java programs running on the remote platform. Loader programs are also provided to load and process the captured data from the VSAM delta cluster (for example, to apply the changes to a database).

Figure 14 provides an overview of how:
1. The VSAM Redirector Connector is used to capture changes in a specific VSAM file on your z/VSE host.
2. A redirector loader program running on a Java workstation then processes this captured data via the *Java-based connector*.

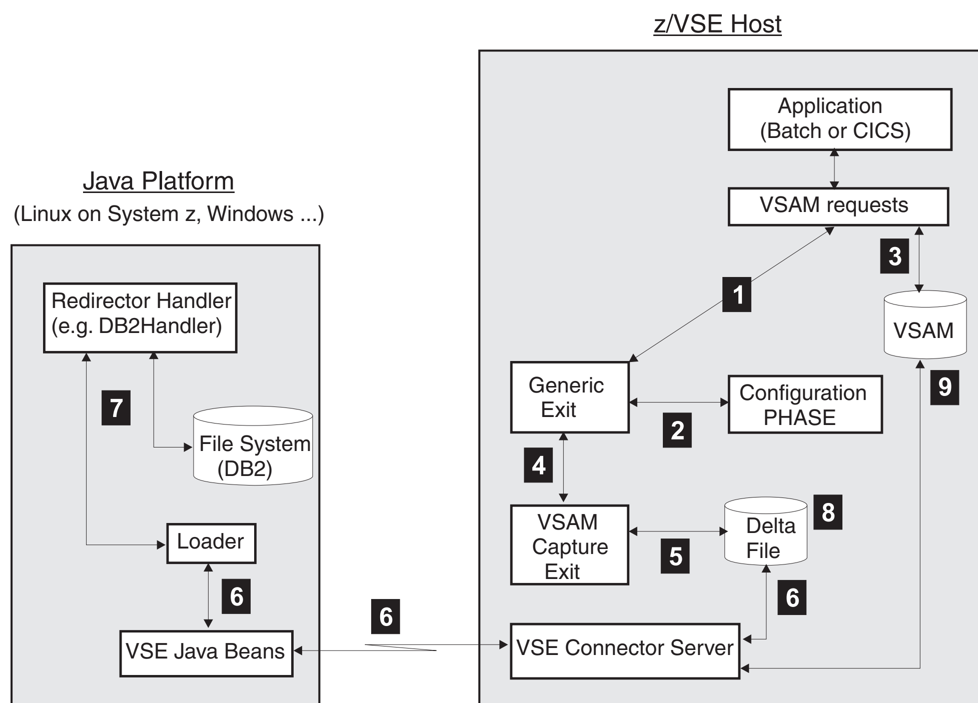


Figure 14. How Asynchronous Data Redirection Is Used

The general processing shown in Figure 13 on page 47 is as follows:

1 An application running on the z/VSE host issues a VSAM command (for example, to open a VSAM file). The request is passed to the *generic exit* (IKQVEX01.PHASE).

2 The generic exit checks whether the VSAM file has been set up to be redirected. To do so, it checks the configuration phase (IESRDCFG.PHASE).

3 If the VSAM file has not been redirected to another Java platform, the generic exit returns and indicates that the VSAM file has not been redirected. It also indicates that the generic exit should not be called again for any request against *this* VSAM file. Normal VSAM processing then continues.

4 Changes (UPDATE, INSERT, or DELETE) made to the VSAM file are captured via the VSAM Capture Exit (IESVSCAP.PHASE).

5 The changes are written to the VSAM delta cluster.

6 The redirector loader reads the VSAM delta cluster.

7 The redirector loader calls the redirector handler, so that it can apply the delta-file changes to the database or file system.

8 When completed, the redirector loader deletes the processed records from the VSAM delta cluster.

9 (Optional). The redirector loader copies the contents of the VSAM file to the remote data base or file system (for example, for initial loading).

VSAM Integration Considerations

For VSAM internal processing (such as the `POINT to END OF FILE`) changes have been made to VSAM so that the VSAM Redirector Client can perform its processing. The original VSAM cluster of a redirected file must, however, still exist on the z/VSE host. It must also contain a dummy record (which you can insert using, for example, the DITTO utility).

In order for all VSAM requests to be redirected, the exit phase (IKQVEX01.PHASE) must return a return code of -1. This indicates to VSAM that no VSAM processing at all is required against this file.

If the exit phase cannot open the configuration phase (IESRDCFG.PHASE), or if a TCP/IP connection is not available, the exit phase reports this to VSAM using two error definitions:

- If the exit phase returns -3 to VSAM, this is converted to a DDNAME NOT FOUND error message, which indicates that the exit phase was unable to connect to the specified Java platform.
- The second error code is -4, converted to a `UNABLE TO CDLOAD` error, which means that the exit phase was unable to load either the VSAM Redirector Client or the configuration phase.

For further details about processing and return-code changes, refer to the online documentation provided with the VSAM Redirector Server (see “Using the Online Documentation Options” on page 26).

Configuring the VSAM Redirector Client / VSAM Capture Exit

The VSAM Redirector Client, VSAM Capture Exit, and related programs are automatically installed on your *z/VSE host* during the installation of z/VSE.

The VSAM Redirector Client and VSAM Capture Exit consist of the following PHASE files:

- IKQVEX01.PHASE (the *generic exit phase*)
- IESREDIR.PHASE (the *VSAM Redirector Client phase*)
- IESVSCAP.PHASE (the *VSAM Capture Exit phase*)
- IESRDCFG.PHASE (the *configuration phase*)
- IESRDANC.PHASE (the *redirector anchor phase*)
- IESRDLDA.PHASE (the *phase used to register a new configuration*)

IKQVEX01 is called each time a VSAM OPEN request is made. Depending on the configuration phase, IKQVEX01 then decides if a VSAM cluster is to be redirected or not:

- If a VSAM cluster *is* to be redirected, IKQVEX01 loads and starts the required exit (for example, IESREDIR or IESVSCAP). You must specify the parameters for these exits in the configuration phase IESRDCFG.PHASE (see "Step 1: Enable the VSAM Redirector Client / VSAM Capture Exit on z/VSE" for details).
- If a VSAM cluster *is not* to be redirected, normal VSAM processing takes place.

An overview of this processing is shown in Figure 13 on page 47.

To configure the VSAM Redirector Client, you must follow these steps:
- "Step 1: Enable the VSAM Redirector Client / VSAM Capture Exit on z/VSE"
- "Step 2: Decide Upon Your Redirection Mode" on page 51
- "Step 3 (Optional): Transfer Your VSAM Data" on page 58
- "Step 4: Create the Configuration Phase" on page 58

Note: Fast Service Upgrade Considerations! If you are performing a Fast Service Upgrade (FSU), you must run the configuration job SKRDCFG to update the IKQVEX01.PHASE in library PRD2.CONFIG.

Step 1: Enable the VSAM Redirector Client / VSAM Capture Exit on z/VSE

The VSAM Redirector Connector is based upon the existing VSAM Data Access (VDA) exit. The VDA exit is represented by the dummy exit phase IKQVEX01.PHASE, which is shipped in library IJSYSRS.SYSLIB.

The VSAM Redirector Client and VSAM Capture Exit also use the VDA exit. Therefore to avoid over-writing any existing changes you have made to IKQVEX01.PHASE, all phases that belong to the VSAM Redirector Client and VSAM Capture Exit are shipped in library PRD1.BASE.

To enable the VSAM Redirector Client or VSAM Capture Exit, you must use skeleton SKRDCFG in Library 59 to configure the VSAM Redirector Client or VSAM Capture Exit. You use this skeleton to

1. Assemble/link the member IESRDCFG.PHASE, and store it in library PRD2.CONFIG.
2. Load the IESRDCFG.PHASE into the SVA (optional).
3. Copy IESVEX01.PHASE to the library PRD2.CONFIG with the name IKQVEX01.PHASE, to activate the exit phase.
4. Load the IKQVEX01.PHASE into the SVA (optional).

   Note: Normally, your original IKQVEX01.PHASE will have been loaded into the SVA. To enable the VSAM Redirector Client, you must replace it in the SVA with the new phase loaded from PRD2.CONFIG. Alternatively, to replace the IKQVEX01 phase you can re-IPL your z/VSE system.

   If you wish to activate a new configuration while the system is running (that is, without restarting your VSAM applications), after completing the steps above you must then:
5. Load the IESRDANC.PHASE into the SVA (this should be done only once per IPL).
6. Execute program IESRDLDA. However, IESRDCFG.PHASE must be *already loaded* into the SVA (as described above). Program IESRDLDA will then activate the new copy of IESRECFG so that it is the current configuration, and the changes become immediately active. However, VSAM files will not be changed

if they are already open. To activate the changes for any VSAM files that are open, you must close and then reopen such files.

After an IPL, you must reactivate the configuration of your VSAM Redirector Client. To do so, you must update USERBG.PROC to:

1. Load the IESRDCFG.PHASE into the SVA.
2. Load the IKQVEX01.PHASE from PRD2.CONFIG into the SVA.
3. Load the IESRDANC.PHASE into the SVA.
4. Execute program IESRDLDA.

The VSAM Redirector Client (phase IESREDIR) uses TCP/IP as its communication protocol. Therefore, any of your applications that you wish to use with the VSAM Redirector Connector must be modified accordingly. You must add the statement:

```
// OPTION SYSPARM='nn'
```

to the JCL for these jobs. The value `'nn'` (the system ID) is contained in the:

```
// EXEC IPNET,SIZE=IPNET,PARM='ID=nn,INIT=... '
```

statement of your TCP/IP startup job.

Note: The value of `'00'` is the default for the system ID. If you accept this default, you are *not* required to add the statement `// OPTION SYSPARM='00'`.

Step 2: Decide Upon Your Redirection Mode

This topic describes the redirection-mode exits provided by IBM.

IBM provides *two* redirection-mode exits:

- The VSAM Redirector Client (IESREDIR), which is the exit for *synchronous* redirection.
- The VSAM Capture Exit (IESVSCAP), which is the exit for *asynchronous* redirection.

Redirection Modes Available for the VSAM Redirector Client

For the *VSAM Redirector Client* (IESREDIR), there are *two* modes you can use:

1. Working with data that resides on another platform.
2. Synchronizing your *existing* VSAM data with data that resides on another platform.

You set the mode of operation using the OWNER parameter within the configuration phase. See “Step 4: Create the Configuration Phase” on page 58 for details.

Notes:

1. For each redirection mode, the VSAM cluster must be defined to the z/VSE system. Therefore, when the VSAM `OPEN` request is executed for this cluster, the VSAM Redirector Client must obtain the following information from the VSAM cluster:
   - Cluster type
   - Key position
   - Key length
   - Maximum record length

2. When you open a redirected file (on a Java platform) for READ processing, the original VSAM cluster must still be defined on the z/VSE host, and must contain at least one 'dummy' record. Otherwise, a VSAM error will occur when the OPEN request is processed.
3. READ requests will not generate any data transfer to the remote Java platform.

Mode 1. Working With Data Residing On Another Platform: If you use this redirection mode (OWNER=REDIRECTOR), your programs that work with VSAM data will never perform any VSAM access operations. All requests are redirected to the VSAM Redirector Client (**iesredir.phase**), which then connects to the VSAM Redirector Server running on a Java platform. The VSAM Redirector Server then performs the request.

Note: You cannot chain exits if any one of these exits has OWNER set to REDIRECTOR.

Figure 15 shows the flow of control when a VSAM PUT is executed.

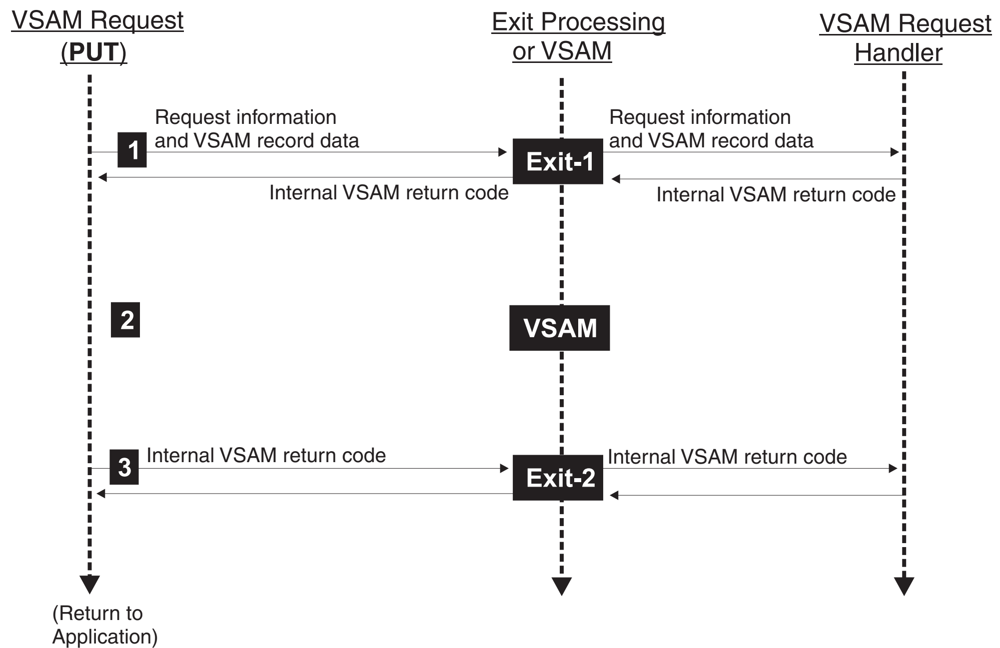


Figure 15. Flow of Control for VSAM PUT Request

1 VSAM calls **Exit-1** *before* VSAM starts to process the request.

2 VSAM does not process any requests.

3 VSAM calls **Exit-2** *after* VSAM has finished processing the request.

Figure 16 on page 53 shows the flow of control when a VSAM GET is executed.

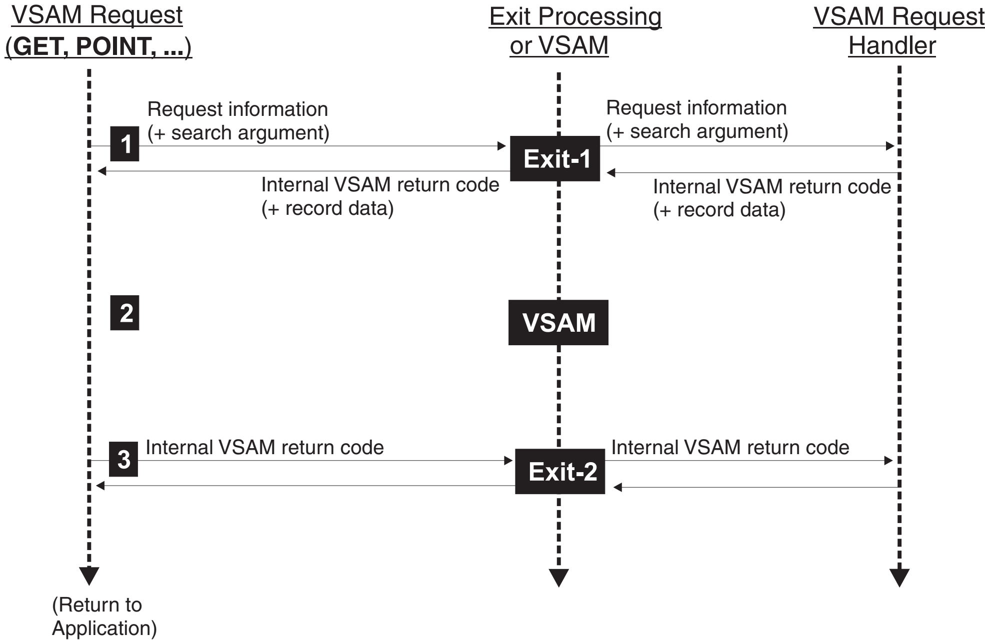


Figure 16. Flow of Control for VSAM GET Request

1 VSAM calls `Exit-1` *before* VSAM starts to process the request.

2 VSAM does not process any requests.

3 VSAM calls `Exit-2` *after* VSAM has finished processing the request.

Mode 2. Synchronizing Your Existing VSAM data: If you use this mode (`OWNER=VSAM`), your programs that work with VSAM data will perform a VSAM access and redirected access. Each VSAM request issues two requests to the VSAM Redirector Client (**iesredir.phase**).

Figure 17 on page 54 shows the flow of control when a VSAM `PUT` is executed.

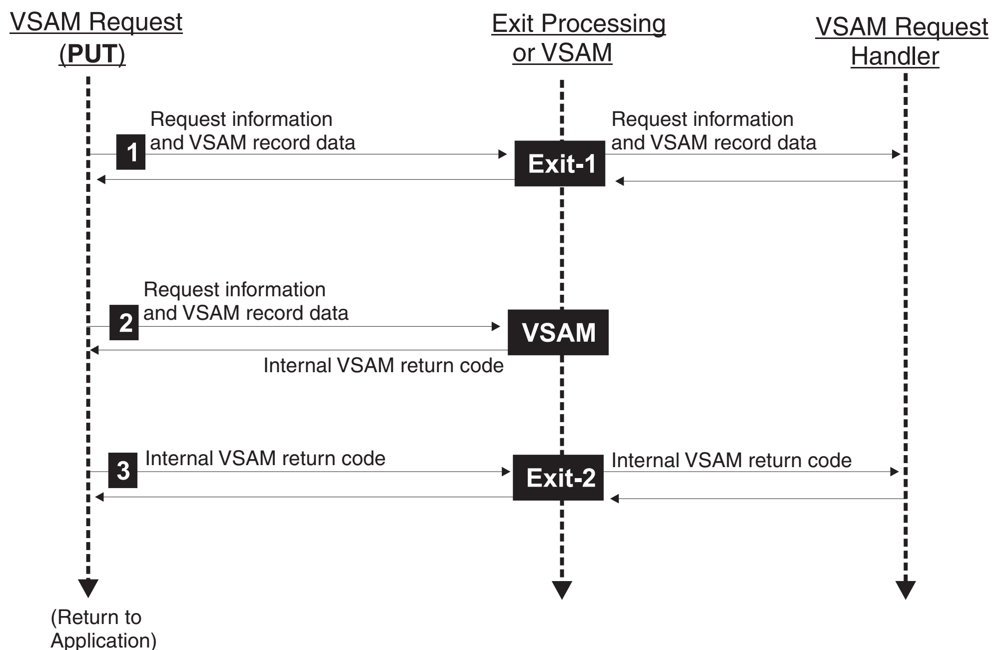


Figure 17. Flow of Control for VSAM PUT Request

1 VSAM calls `Exit-1` *before* VSAM starts to process the request.

2 VSAM processes the request.

3 VSAM calls `Exit-2` *after* VSAM has finished processing the request.

Figure 18 shows the flow of control when a VSAM `GET` is executed.

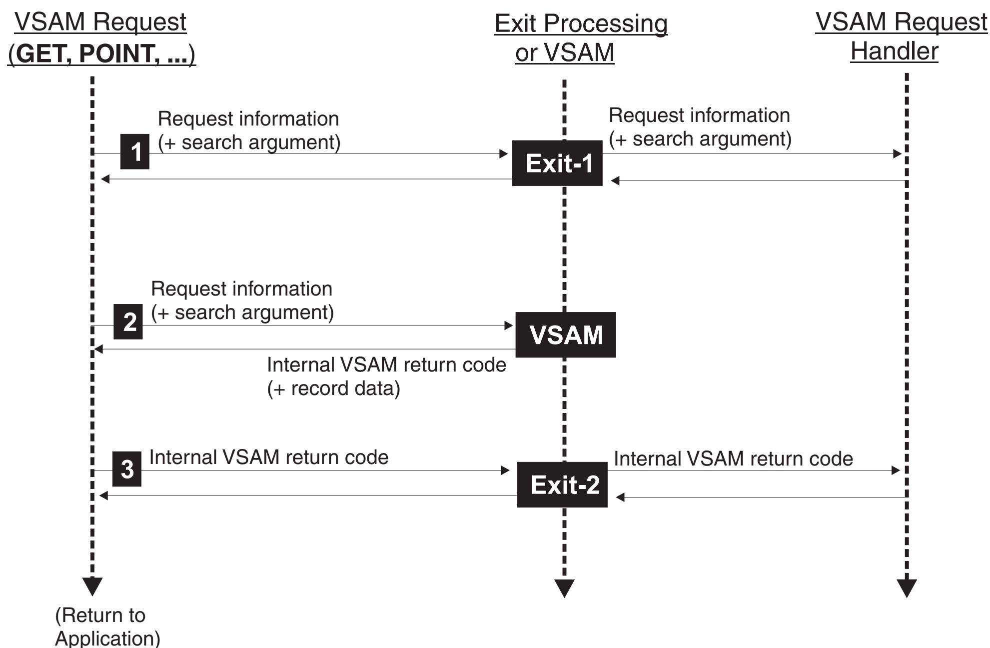


Figure 18. Flow of Control for VSAM GET Request

1 VSAM calls `Exit-1` *before* VSAM starts to process the request.

2 VSAM processes the request.

3 VSAM calls **Exit-2** *after* VSAM has finished processing the request.

Redirection Modes Available for the VSAM Capture Exit

For the *VSAM Capture Exit* (IESVSCAP), there are *three* modes you can use:
- WebSphere MQ
- VSAM delta cluster
- Local processing

Using the WebSphere MQ Redirection Mode: When using *WebSphere MQ*, you can use *two* modes:
- When using the *MQ Server mode*, MQ Series for VSE V2.1 or later must be installed and running on your z/VSE host. The delta message is created and put into a MQ queue on z/VSE. The MQ queue can be a remote queue, which means the messages are transferred asynchronously by a MQ channel to another MQ queue on another system.
- When using *MQ Client mode*, the *WebSphere MQ Client for VSE* must be installed and running on z/VSE. The delta message is created and put directly into a MQ queue on a remote MQ Server.

Using the VSAM Delta Cluster Mode: When using a VSAM delta cluster, you can use *two* modes:
- *Cumulative mode* uses a KSDS type delta cluster. The key of the VSAM delta cluster is the same as the original cluster, or the RRN/RBA. This means, the offset is adjusted to cover the *delta header* (see “Understanding the Layout of Delta Records/Messages” on page 56). Therefore, a record can be stored only once in the VSAM delta cluster. This means, only the last change of the record is reflected in the VSAM delta cluster. However, this is enough information to synchronize a database.
- *Journaling mode* uses a KSDS or ESDS type VSAM delta cluster. Whenever a change is to be recorded into the VSAM delta cluster, a new record is added using the TOD Clock as key (in case of KSDS). Therefore, since the TOD clock is unique, every change of a record is reflected in the VSAM delta cluster (including the order of changes).

Both modes can be used to synchronize a database. However, some circumstances might require applying the changes of a record in its original order. In this case, the journaling mode must be used.

Using the Local Processing Mode: When performing local processing, you must specify an exit that implements your own logic to handle the delta record.

For all three VSAM Capture Exit redirection modes (WebSphere MQ, VSAM delta cluster, and local processing), you can specify an additional *decision exit*:
- The decision exit is called whenever a delta record/message is to be written.
- The decision exit can decide if the delta record/message should be written or ignored. Such a decision can be based on the contents of the record, e.g. when changes in some particular fields of the record are not of interest.
- The decision exit gets the old and the new contents of the record (only for UPDATE).
- For local processing mode, the decision exit also implements the logic to process the delta record.

For an example of how to code a decision exit, see skeleton SKDECEXT in ICCF Library 59.

Understanding the Layout of Delta Records/Messages

The VSAM Capture Exit creates delta records that each start with a *delta header* of length 38 or 42 bytes.

```
TODClock        8 bytes     TOD Clock value of update (TOD = Time of day)
JobName         8 bytes     Job Name of Program
Phase           8 bytes     Phase name of program
Origin          8 bytes     Origin value, e.g. Label name
PartID          2 bytes     Partition ID
OpCode          1 byte      'I'=insert,'U'=update,'D'=delete
Flags           1 byte      '01'X= RRD/RBA follows
RecordLen       2 byte      Length of Record (exl. Header and RBA/RRN)
```

For ESDS, RRDS, or VRDS datasets, a 4-byte RRN/RBA then follows the capture header:

```
RBA/RRN         4 bytes     RBA or RelRecNo of record
```

The remainder of the delta record is filled with a copy of the *original record*. The actual length of the delta record is the length of the original record plus 38 or 42 bytes.

Using a VSAM Delta Cluster

If the VSAM Capture Exit is used with a VSAM delta cluster, you must define the VSAM delta cluster as follows:

- For Cumulative Mode -

```
Original Cluster is a KSDS:
  Cluster Type = KSDS
  MaxRecLen    = Original MaxRecLen + 38 or larger
  KeyPos       = Original KeyPos + 38
  KeyLen       = Original Key Len
Original Cluster is a ESDS:
  Cluster Type = KSDS
  MaxRecLen    = Original MaxRecLen + 42 or larger
  KeyPos       = 38
  KeyLen       = 4
Original Cluster is a RRDS/VRDS:
  Cluster Type = KSDS
  MaxRecLen    = Original MaxRecLen + 42 or larger
  KeyPos       = 38
  KeyLen       = 4
```

- For Journaling Mode -

```
Original Cluster is a KSDS:
  Cluster Type = KSDS or ESDS
  MaxRecLen    = Original MaxRecLen + 38 or larger
  KeyPos       = 0 (applies only if delta cluster is KSDS)
  KeyLen       = 8 (applies only if delta cluster is KSDS)
Original Cluster is a ESDS:
  Cluster Type = KSDS or ESDS
  MaxRecLen    = Original MaxRecLen + 42 or larger
  KeyPos       = 0 (applies only if delta cluster is KSDS)
  KeyLen       = 8 (applies only if delta cluster is KSDS)
Original Cluster is a RRDS/VRDS:
  Cluster Type = KSDS or ESDS
  MaxRecLen    = Original MaxRecLen + 42 or larger
  KeyPos       = 0 (applies only if delta cluster is KSDS)
  KeyLen       = 8 (applies only if delta cluster is KSDS)
```

It may be useful to define the VSAM delta cluster as REUSABLE. This makes it easier to clear the VSAM delta cluster, after the changes have been transferred and loaded into the database.

For journaling mode, you can use one VSAM delta cluster for capturing the changes of *several* VSAM files. To do so:

- You must ensure that the maximum record length of the VSAM delta cluster is large enough to hold the largest record (plus header length) of the captured clusters.
- It might also be useful to specify a unique ORIGIN name in the configuration to be able to distinguish delta records from different VSAM clusters later on.
- You should usually define the VSAM delta cluster with share options 4, that is SHR(4,3), to permit concurrent access from several partitions.

The VSAM delta cluster cannot be defined using option SPEED. You should use RECOVERY instead. If the VSAM Capture Exit detects that the VSAM delta cluster is using SPEED option, it will reject the OPEN request and return an appropriate message.

With journaling mode, the VSAM delta cluster can be either a KSDS or an ESDS cluster. A KSDS VSAM delta cluster gives you the possibility to delete delta records when they have been processed. In addition, KSDS can be defined as XXL (extra large) to get around the 4 GB limit.

Using WebSphere MQ

When using mode MQServer or MQClient, you must specify the MQ server name. You can specify the MQ server name using either:

- The SETPARM statement -

```
// SETPARM MQBISRV=servername  (default is MQBISERV)
```

- The configuration entry (MQISRV parameter), as described in “Parameters When MODE=MQSERVER and MODE=MQCLIENT” on page 61.

If the capture exit is used with WebSphere MQ, the MQ queue has to be defined in order to hold the maximum record length of a VSAM delta cluster (as described previously).

Guidelines for Using WebSphere MQ:

- If you use mode MQServer, you must specify the name of the MQ Server (the default is MQBISERV).
- If you use mode MQClient, you must specify the name of the MQ Client Bridge (the default is MQBISERV).
- If both the MQ Server and MQ Client Bridge are running on the same system, you must use different names for the server and bridge.
- Please ensure you add the WebSphere MQ and/or WebSphere MQ Client library to the LIBDEF for all involved partitions.

You must also fulfill the following prerequisites:

- MQSeries® for VSE Version 2.1.2 or later must be installed and configured to use MQServer Mode. To use MQClient mode, the WebSphere MQ Client package must be installed. You can download file **mqc5.zip** free-of-charge from this address:

  `http://www.ibm.com/systems/z/os/zvse/downloads/`

- You must use a *separate* CICS region to run WebSphere MQ.

  Note: If you capture changes made by a program that runs in the CICS partition that is also used to run WebSphere MQ, **this will create a deadlock!** This is true for *both* modes (WebSphere MQ and MQClient)!

- If you use MQ Client, the MQ Client Bridge must be started in CICS (MQCI transaction). For further information about the MQ Client Bridge, refer to the readme file provided with the MQ Client package.
- You must define the MQ Client programs and transactions in CICS. Please refer to sample member MQCICSD.Z contained in the MQ Client package.

Step 3 (Optional): Transfer Your VSAM Data

To use your existing VSAM datasets with the VSAM Redirector Connector, you can migrate your VSAM data to the file system you require (for example, DB2 format). For details of how to map VSAM data to a relational structure suitable for DB2 processing, either:

- Refer to Chapter 12, “Mapping VSE/VSAM Data to a Relational Structure,” on page 103.
- Use the Graphical User Interface (Mapper Config GUI).

To transfer your data from the z/VSE host to your target Java platform's file system, you can use either:

- The IDCAMS REPRO utility:
  1. Define a VSAM cluster (that has the same properties as the source cluster) as the target for your redirection / transfer process.
  2. Change the configuration phase, so that your VSAM cluster is redirected (`OWNER=REDIRECTOR`). For details, see “Step 4: Create the Configuration Phase.”
  3. Ensure that the VSAM Redirector Server, and the handler you have defined in the configuration phase, are both running on your target Java platform.
  4. Copy the data into your redirected VSAM cluster, using the IDCAMS REPRO utility. After completing this action, your data will now be stored on the target Java platform's file system.
- A redirector loader utility:
  1. Setup the redirector-loader configuration file as described in the online documentation provided with the VSAM Redirector Server (see “Using the Online Documentation Options” on page 26).
  2. Run the redirector loader.

Step 4: Create the Configuration Phase

VSE library 59 contains an example skeleton SKRDCFG which you can use to create a configuration phase. “Example of Job to Create the Configuration Phase” on page 62 shows the sample job skeleton SKRDCFG. Below are the parameters you set in this job.

Mandatory Parameters

CATALOG=

VSAM catalog name of the file to be redirected. You can use the wildcard * in your parameter definition. In this case, you must also set `CLUSTER=*`. However, be aware that if you use `CATALOG=*` this will redirect *all* catalogs (and clusters contained in those catalogs).

Notes:

1. If the master catalog is redirected, **you might not be able to startup your z/VSE system!**
2. Entries that contain wildcards will only used providing no other matching entry can be found.

CLUSTER=

VSAM cluster name. You can use the wildcard * in your parameter

definition. However, be aware that if you use `CLUSTER=*` this will redirect *all* clusters belonging to the specified catalog.

Name of the exit phase to use.
- If you specify an IBM-provided exit (IESREDIR or IESVSCAP), the parameters listed below apply.
- If you specify another value for `EXIT` (for example, a vendor-provided exit), no further parameters apply.

Optional Parameters

Using these optional parameters, you can specify additional filters when specifying the clusters that are to be redirected.

CATDD=
The label name of the catalog. The default is CATDD='*'. If you enter a value for CATDD (other than the default), both the CATALOG and CATDD will be used for checking if the file is to be redirected.

CLUDD=
The label name of the cluster. The default is CLUDD='*'. If you specify a value for CLUDD (other than the default), both the CLUSTER and CLUDD are used to check if the file is to be redirected.

MSG= Can take one of the following values:
YES (Default) When a redirected VSAM cluster is opened, message IESC2009I will be displayed upon the Console.
NO When a redirected VSAM cluster is opened, message IESC2009I will be suppressed.

PART=
The partition ID (for example F4) of the partition from which redirection is *only* possible. The default value is PART='*'.

NOTPART=
The partition ID (for example F4) of the partition from which redirection is *not* possible. The default is that *all* partitions are available for redirection.

Parameters When Using the IESREDIR Exit

OWNER=
Can take one of these values:
REDIRECTOR
All requests are redirected to the VSAM Redirector Client (**IESREDIR.PHASE**), which then connects to the VSAM Redirector Server running on a Java platform. The VSAM Redirector Server then performs the request.
VSAM
Dual processing occurs (both VSAM processing and redirecting requests to the VSAM Redirector Client).

PROTOCOL=31
This parameter can be used with z/VSE 4.1 or later only. This parameter instructs the VSAM Redirector Client to switch back to an older version of the protocol. It enables you to run a z/VSE 4.1 host with an older version of the VSAM Redirector Server on the remote side. This option is intended to provide a smooth migration when upgrading your VSE system.

IP= A mandatory parameter. The IP address of the server where the VSAM Redirector Server is installed, and to which a connection is to be made.

PORT=
(Optional). The port number of the server where the VSAM Redirector Server is installed, and to which a connection is to be made. The default port number used with the VSAM Redirector Server is **2387**, which has been assigned by the Internet Assigned Numbers Authority (IANA).

HANDLER=
A mandatory parameter. The name of the Java class to be started, which represents the redirector handler to be used with this configuration entry.

OPTIONS=
A mandatory parameter. A string containing data to be transferred to the redirector handler, as an option string. You can insert your own settings here. In the example shown in “Step 4: Create the Configuration Phase” on page 58, this string contains information required by the IBM-supplied redirector handlers *DB2Handler*, *DBHandler*, and *CSVFileHandler* (DB/2 system, username, password, and so on). If you specify a blank value for this parameter (‘ ’), the redirector handler will receive blanks as the option string.

PROTOCOL=
(Optional). The default is to use the z/VSE 4.1 version of the protocol. To use the z/VSE 3.1 version of the protocol, you must set the parameter to **31**.

Parameters When OWNER=VSAM

IGNOREERROR=
An optional parameter, whose default is `IGNOREERROR=NO`. If you set `IGNOREERROR=YES`, a `VSAM OPEN` request will not return an error, even if the VSAM Redirector Server cannot be accessed.

PUTREQONLY=
An optional parameter, whose default is `PUTREQONLY=YES`. If you set `PUTREQONLY=YES`, only `INSERT`, `UPDATE`, and `DELETE` requests will be redirected to the VSAM Redirector Server. You might find this parameter useful if you want to collect statistics about your redirector handler, excluding requests such as `POINT`, `GET`, and so on.

Parameters When Using the IESVSCAP Exit

MODE=
Can take one of these values:
- **JOURNALING** or **CUMULATIVE** for use with a VSAM delta cluster.
- **MQSERIES** or **MQCLIENT** for use with WebSphere MQ.
- **LOCAL** for local processing.

ORIGIN=
Specifies a name of up to 8 characters length. This name can be freely chosen. It is part of the delta header, and therefore will appear in each delta record in the delta cluster or MQ message. It can later be used to distinguish delta records/messages from different clusters (their origins). If ORIGIN is omitted, the current DLBL name of the cluster is used.

DECEXIT=
Name of the phase that implements the decision exit. It is optional except for MODE=LOCAL. For a programming example, see skeleton SKDECEXT in ICCF Library 59.

IGNOREERROR= NO | YES
(Optional). If set to YES, all errors will be ignored and processing will continue despite the occurrence of an error. Please note that this may cause data inconsistencies, since not all changes will be contained in the delta cluster or queue.

Parameters When MODE=JOURNALING or MODE=CUMULATIVE

DELTADD=
Specifies the DLBL name of the delta cluster that is to be used to store delta records. This parameter is *only* permitted when using journaling or cumulative mode.

DELTACAT=
(Optional). Specifies the 44-byte file-id of the catalog in which the delta cluster resides. This parameter is *only* permitted when DELTADD is *not* specified. It must be specified together with DELTACLU.

DELTACLU=
(Optional). Specifies the 44-byte file-id of the delta cluster to be used to store delta records. This parameter is *only* permitted when DELTADD is not specified. It must be specified together with DELTACAT.

SHARE=NONE | ENDREQ | TCLOSE
(Optional). Specifies whether an ENDREQ or TCLOSE request is to be issued after each delta record insertion.

DSNSTR=
An optional parameter (default is 255). Specifies the number of strings between 0 and 255 that are to be used with dataset name sharing. 0 means no dataset name sharing is used.

DELTATYPE=KSDS | ESDS
(Optional). ESDS is only permitted when MODE=JOURNALING.

Parameters When MODE=MQSERVER and MODE=MQCLIENT

QMGR=
Name of the WebSphere MQ queue manager (up to 48 characters).

QNAME=
Name of the WebSphere MQ queue (up to 48 characters).

MQISRV=
XPCC name of the z/VSE MQ Service. See “Parameters When MODE=MQCLIENT” for more information.

Parameters When MODE=MQCLIENT

MQSERVER=
IP or hostname of MQServer (up to 100 characters).

MQCHANNEL=
Channel name used with MQServer (up to 20 characters).

Example of Job to Create the Configuration Phase

The example below shows the sample job skeleton SKRDCFG.

Figure 19. Job to Produce a Configuration Phase for the VSAM Redirector Connector

```
* $$ JOB JNM=RDCONFIG,CLASS=A,DISP=D
// JOB RDCONFIG GENERATE REDIRECTOR CONFIG PHASE
* ********************************************************************
* STEP 1: ASSEMBLE AND LINK THE CONFIG TABLE                         *
* ********************************************************************
// LIBDEF *,CATALOG=PRD2.CONFIG
// LIBDEF *,SEARCH=PRD1.BASE
// OPTION ERRS,SXREF,SYM,NODECK,CATAL,LISTX
   PHASE IESRDCFG,*,SVA
// EXEC ASMA90,SIZE=(ASMA90,64K),PARM='EXIT(LIBEXIT(EDECKXIT)),SIZE(MAXC
               -200K,ABOVE)'
IESRDCFG CSECT
IESRDCFG AMODE ANY
IESRDCFG RMODE ANY
*
         IESRDENT CATALOG='VSESP.USER.CATALOG',                        X
               CLUSTER='MY.TEST.CLUSTER1',                             X
               EXIT='IESREDIR',                                        X
               OWNER=REDIRECTOR,                                       X
               IP='10.0.0.1',                                          X
               HANDLER='com.ibm.vse.db2handler.DB2Handler',            X
               OPTIONS='db2url=jdbc:db2:redir;db2user=hugo;            X
                        db2password=hugospw;db2table=mydata'
*
          IESRDENT CATALOG='VSESP.USER.CATALOG',                       X
               CLUSTER='MY.TEST.CLUSTER2',                             X
               EXIT='IESVSCAP',                                        X
               MODE=JOURNALING,                                        X
               DELTADD='DELTAFI',                                      X
               DELTATYPE=KSDS,                                         X
               SHARE=ENDREQ,                                           X
               ORIGIN='TEST2'
*
         IESRDENT CATALOG='VSESP.USER.CATALOG',                        X
               CLUSTER='MY.TEST.CLUSTER3',                             X
               EXIT='IESVSCAP',                                        X
               MODE=MQSERVER,                                          X
               QMGR='VSE.QUEUE.MANAGER',                               X
               QNAME='CAPTURE.INPUT.QUEUE',                            X
               ORIGIN='TEST3'

         IESRDENT CATALOG='VSESP.USER.CATALOG',                        X
               CLUSTER='MY.TEST.CLUSTER2',                             X
               EXIT='VENDOREX'
*
         END
/*
// IF $MRC GT 4 THEN
// GOTO NOLINK
// EXEC LNKEDT,PARM='MSHP'
/. NOLINK
/*
* ********************************************************************
* STEP 2: LOAD THE IESRDCFG.PHASE INTO THE SVA (OPTIONAL)            *
* ********************************************************************
* LIBDEF *,SEARCH=PRD2.CONFIG
* SET SDL
* IESRDCFG,SVA
* /*
* ********************************************************************
* STEP 3: COPY IESVEX01.PHASE INTO PRD2.CONFIG AS IKQVEX01           *
* ********************************************************************
// EXEC LIBR,PARM='MSHP'
  CONNECT S=PRD1.BASE:PRD2.CONFIG
```

```
  COPY IESVEX01.PHASE:IKQVEX01.PHASE REPLACE=YES
/*
* *********************************************************************
* STEP 4: LOAD THE IKQVEX01.PHASE INTO THE SVA (OPTIONAL)            *
* *********************************************************************
* LIBDEF *,SEARCH=PRD2.CONFIG
* SET SDL
* IKQVEX01,SVA
* /*
* *********************************************************************
* STEP 5: LOAD THE IESRDANC.PHASE INTO THE SVA (OPTIONAL)            *
*         THIS SHOULD BE DONE ONLY ONCE  !!                          *
* *********************************************************************
* // LIBDEF *,SEARCH=PRD2.CONFIG
* SET SDL
* IESRDANC,SVA
* /*
* *********************************************************************
* STEP 6: REGISTER THE CURRENT CONFIGURATION PHASE                   *
* *********************************************************************
* // LIBDEF *,SEARCH=PRD1.BASE
* // EXEC IESRDLDA
* /*
/&
* $$ EOJ
```

Installing the VSAM Redirector Server

This topic describes the activities you must perform on each Java platform where you plan to install migrated datasets.

The main activities that you must perform on each Java platform where you plan to install migrated datasets, are:
- “Step 1: Download the Install-File and Perform the Installation”
- “Step 2: Configure the Properties File” on page 65
- “Step 3: Implement a VSAM Redirector Handler” on page 65

Step 1: Download the Install-File and Perform the Installation

You install the VSAM Redirector Server on a Java-enabled platform.

Note: Before you begin, you must already have installed the Java Development Kit (JDK) 1.5 or later on the development platform where you plan to install the VSAM Redirector Server. If you do not have JDK 1.5 or later installed, refer to “Installing and Configuring Java” on page 19 for details of how to install it.

Step 1.1: Obtain a Copy of the VSAM Redirector Server

To obtain a copy of the VSAM Redirector Server, you must decide if you wish to obtain it:
- From the Internet.
- By installing the VSE Connectors Workstation Code component from the Extended Base Tape.

To obtain the VSAM Redirector Server *from the Internet*, you should:
1. Start your Web browser and go to URL:

   `http://www.ibm.com/systems/z/os/zvse/downloads/`
2. From within the VSAM Redirector Server section, download the file **redir*nnn*.zip** to the directory where you wish to install the VSAM Redirector Server. **Note:** *nnn* refers to the current VSE version (for example, **redir430.zip**).

To obtain the VSAM Redirector Server *by installing the VSE Connectors Workstation Code component*, you should:

1. Install the VSE Connectors Workstation Code component from the Extended Base Tape. After you have installed this component, the VSAM Redirector Server W-book **iesvsmrd.w** will be stored in z/VSE sublibrary PRD2.PROD.
2. Use the FTP (file transfer program) utility of TCP/IP for VSE/ESA to download **iesvsmrd.w** to the directory where you wish to install the VSAM Redirector Server.

Notes:

1. You must download **iesvsmrd.w** in *binary*.
2. Make sure that Unix mode is ***turned off***. Otherwise **iesvsmrd.w** will be downloaded in ASCII mode, even when you specify *binary*. *Unix mode* is one parameter of your VSE FTP daemon. Some FTP clients might *force* Unix mode to be turned on! The example below shows how a successful transfer of **iesvsmrd.w** was made using a batch FTP client. The place where the UNIX mode is set, is shown as bold.

```
c:\temp>ftp 9.164.155.2
Connected to 9.164.155.2.
220-TCP/IP for VSE -- Version 01.05.F  -- FTP Daemon
    Copyright (c) 1995,2006 Connectivity Systems Incorporated
220 Service ready for new user.
User (9.164.155.2:(none)): sysa
331 User name okay, need password.
Password:
230 User logged in, proceed.
ftp> cd prd2
250 Requested file action okay, completed.
ftp> cd prod
250 Requested file action okay, completed.
ftp> binary
200 Command okay.
ftp> get iesvsmrd.w
200 Command okay.
150-File: PRD2.PROD.IESVSMRD.W
    Type: Binary Recfm: FB Lrecl:    80 Blksize:      80
    CC=ON  UNIX=OFF RECLF=OFF TRCC=OFF CRLF=ON  NAT=NO
150 File status okay; about to open data connection
226-Bytes sent:       4,756,400
    Records sent:        59,455
    Transfer Seconds:        16.52 (    290K/Sec)
    File I/O Seconds:         3.94 (  1,548K/Sec)
226 Closing data connection.
4756400 bytes received in 17,12 seconds (277,91 Kbytes/sec)
ftp> bye
221 Service closing control connection.
c:\temp>ren iesvsmrd.w redir.zip
```

Step 1.2: Perform the Installation of the VSAM Redirector Server

To perform the installation of the VSAM Redirector Server, you must:

1. Unzip the file **redir.zip**, which contains these files:
   - setup.jar (contains the VSAM Redirector Server code)
   - setup.bat (an install batch file for Windows)
   - setup.cmd (an install batch file for OS/2)
   - setup.sh (an install script for Linux/Unix)
2. Start the batch file (by double-clicking the file) that is applicable to your operating-system platform.
3. The installation process now begins, and you are guided through various installation menus.

4. To access the HTML-based documentation, you can now use your Web browser to open the file **... (*Redirector root directory*)/doc/index.html**.

Step 2: Configure the Properties File

The properties file for the VSAM Redirector Server is called **VSAMRedirectorServer.properties**, which is a text file that you can edit using any text editor.

Comment lines begin with a # character in the first column.

These are the settings that you define in **VSAMRedirectorServer.properties**:

messages= on|off
: If you define **messages= on**, all messages will be printed. If you define **messages= off**, messages will not printed (and "quiet mode" will be active).

listenport = *TCP/IP portnumber*
: Port number which the VSAM Redirector Server uses to listen for requests.

maxconnections = *number*
: Maximum number of connections that are allowed from VSAM Redirector Clients.

codepagetranslator = com.ibm.vse.server.DefaultTranslator
: Codepage translator class to be used for converting Strings from:
  - EBCDIC into ASCII
  - ASCII into EBCDIC

  The IBM-supplied default is shown above.

Step 3: Implement a VSAM Redirector Handler

VSAM Redirector Handlers (referred to simply as *redirector handlers*) are used by the VSAM Redirector Server.

Redirector handlers are shown in Figure 13 on page 47, and are programmed in Java. This topic describes how redirector handlers are implemented, under these headings:
- "Coding VSAM Logic and Parameters"
- "Calling a VSAM Redirector Handler" on page 66
- "Error Reporting" on page 66
- "Datatype Conversions" on page 67
- "Getting a Map Dynamically Into Your Redirector Handler" on page 67
- "IBM-Supplied Example of DB2-Related Handler" on page 70

Coding VSAM Logic and Parameters

To redirect VSAM data into other data formats (SQL database, flat file, and so on) so that this data is transparent to existing applications, you must simulate all VSAM behavior in your redirector handler. Therefore you must include positioning and error-reporting logic in your redirector handler.

You can program your own redirector handlers and include them for use by the VSAM Redirector Connector. You must also code an interface which your redirector handler provides, by implementing *case methods* to handle:
- OPEN requests
- CLOSE requests
- Record requests, which are:
  - GET

- PUT (UPDATE+INSERT)
- ENDRQ
- POINT

If you wish to write a redirector handler that is to copy VSAM files (in the same way as the IBM-supplied *DB2Handler*), your redirector handler must:

1. Copy the VSAM logic.
2. Respond in the same way as VSAM itself would.

For details of how to code VSAM logic and parameters, refer to the:

- *VSE/VSAM User's Guide and Application Programming*, SC33-8316
- *VSE/VSAM Commands*, SC33-8315
- Javadoc for *VSAMRequestInfo.java* and *VSAMFileInfo.java*.

Calling a VSAM Redirector Handler

The VSAM Redirector Server is implemented in Java and is not delivered with source code. The VSAM Redirector Server can be started after the:

1. properties file has been configured,
2. redirector handler code has been copied into the directory where the VSAM Redirector Server is installed.

When the VSAM Redirector Server has been started:

1. The VSAM Redirector Server reads the properties file.
2. The VSAM Redirector Server listens for VSAM Redirector Clients on the TCP/IP port that you defined in "Step 2: Configure the Properties File" on page 65.
3. When the VSAM Redirector Server receives an OPEN request:
   a. The redirector handler that was defined in configuration.phase, is instantiated.
   b. The redirector handler's OPEN method is called.
4. The redirector handler terminates in the way that is defined by the *CLOSE method* call.

Error Reporting

Redirector handlers report an error by producing an *Exception*. The VSAM Redirector Server intercepts the Exception and sends the error code to the VSAM Redirector Client running on the z/VSE host. *No data is sent together with the error code*.

Error codes that are sent to the VSAM Redirector Client:

- must be an internal VSAM return code
- are returned to VSAM into register 15 (R15).

A new error code has been created for use with the VSAM Redirector Connector: when a redirector handler sends a `DUPLICATE RECORD` warning to a VSAM application (when more than one record is found in alternate index access/path access and when other records follow) it produces an Exception. However, in this case the record data is not sent! To return this warning message, you must therefore use error code 255 (which will be converted in the server-part to the required warning message).

Also refer to "VSAM Integration Considerations" on page 49 for further error-reporting considerations.

Datatype Conversions

Record data that is transferred from the VSAM Redirector Client running on the z/VSE host, consists of EBCDIC characters. Therefore, all data that is interpreted as Strings must be converted to *ASCII characters*.

From z/VSE 4.1 onwards, these IBM-provided *converters* are available for performing datatype conversions.

- BINARY - Binary data, no translation
- BIT - a single bit in a byte
- DATETIME - converter for various date and time formats
- FIXEDTEXTNUMBER - fixed point numbers stored in a textual format
- FLOAT - floating point numbers (BFP and HFP format)
- FLOATTEXTNUMBER - floating point numbers stored in a textual format
- INTEGER - integer numbers
- PACKED - packed decimal numbers
- S2Y - special format used for Year 2000 convertions
- STRING - textual strings
- TOD - TOD Clock (Time of Day) converter
- ZONED - zoned decimal numbers

Notes:

1. The IBM-provided redirector handler DB2Handler (see "Using the IBM-Provided VSAM Redirector Handlers") uses these converters.
2. DBHandler uses these converters for data-type translation.
3. For further details about converters, refer to the online documentation.

Getting a Map Dynamically Into Your Redirector Handler

These are the ways in which you can get a map into your redirector handler *dynamically*:

- Define the map as an option string in the **config.phase** and parse it.
- Use the z/VSE Connector framework to get a VSAM map from the cluster.
- Use an XML file and then parse it (you can create such a file using the *MapTool* – see *CreateDB2Tables.java* for an example of how to do so).
- Store the map in another location (for example, in a database table – see *DB2Handler.java* or *DBHandler.java* for examples of how to do so).

Using the IBM-Provided VSAM Redirector Handlers

z/VSE provides these VSAM Redirector Handlers:

- **DB2Handler**
- **DBHandler**
- **CSVFileHandler**

Table 3 provides a summary of the characteristics of these three handlers.

Table 3. Currently-Supplied VSAM Redirector Handlers

Function	DB2Handler	DBHandler and CSVFileHandler (z/VSE 4.1 and later)
Modes of redirection	• owner=VSAM • **owner=REDIRECTOR**	• **owner=VSAM**

Table 3. Currently-Supplied VSAM Redirector Handlers (continued)

Function	DB2Handler	DBHandler and CSVFileHandler (z/VSE 4.1 and later)
Supported VSAM requests	**For KSDS, ESDS:** **GET**, **POINT**, INSERT, UPDATE, DELETE, ENDRQ	**DBHandler for KSDS:** INSERT, UPDATE, DELETE, ENDRQ **DBHandler for ESDS,VRDS,RRDS:** INSERT, ENDRQ **CSVFileHandler for KSDS,ESDS,RRDS,VRDS:** INSERT, ENDRQ
Targets types for the data	SQL database system	SQL database system, CSV files
Targets for the data	One database table per VSAM file	Multiple tables per VSAM file
Mapping of VSAM key	Key must be 1 field of type STRING	Support for multiple data fields in key
Support for fixed length lists	Each list entry must be one field in database table	Support for normalized tables, each list entry can be stored in a list entry table
Support for variable length lists	n/a	Supported
Support for record types	n/a	Support for different mappings depending on the value of a field in record

Additional Information to Table 3 on page 67:

- The *DB2Handler* redirector handler allows you to map *all* data fields belonging to a VSAM record to the fields into *one* SQL database table. The VSAM key field must be mapped as one complete field with the type STRING. This is required in order to support a VSAM GET request, because the handler must:
  - Locate the VSAM key that the application requested.
  - Be able to search forwards and backwards through the VSAM keys.
- The limitation with using the *DB2Handler* is, that the one-to-one mapping of all data base belonging to a VSAM record into *one* SQL database table often results in a unpractical database table design. For example, if a VSAM record contains the data `MYLIST PIC X(20) OCCURS 10`, the database table will contain 10 fields with names such as `MYLIST1, MYLIST2, ....`
- Using the *DBHandler* and *CSVFileHandler* redirector handlers:
  - The mapping limitation described above is removed. The database table design for these two handlers is such that a list such as `MYLIST PIC X(20) OCCURS 10` will be *normalized*. This is done by creating another database table that has the same key as the record in the main table. This database table contains *one* record for *each* entry in the list. Since each record in the database table includes the key of the main data record, these records will be linked to the main record. As a result, a different count of list records for each main data record can be made, without the need to change the database table.
  - You can map different types of records within the same VSAM file to *different* database tables. This mapping is based upon the value of a data field in a VSAM record.
  - Your applications that use VSAM *cannot* retrieve data (using a VSAM GET) from the database tables. The two redirector handlers can be used to (1) synchronize VSAM data with data contained in the database, or (2) migrate data from a VSAM file into a database.

– For further details about these two redirector handlers, refer to the online documentation provided with the VSAM Redirector Server (see "Using the Online Documentation Options" on page 26).

Using the IBM-Provided VSAM Redirector Loaders

The IBM-provided *VSAM Redirector Loaders* (referred to as *redirector loaders*) allow you to download data from VSAM clusters. The transfer mechanism has been optimized for the mass-transfer of data.

All the IBM-provided redirector loaders use VSAM Redirector Handlers (referred to as *redirector handlers*) to process the downloaded data. You do not need to code the logic to process the data (for example, to insert the data into a database). Instead, a redirector loader can:
1. Call a configured redirector handler.
2. Let the redirector handler process the data.

Usually, the redirector loader calls the redirector handler with an INSERT request. This allows you to use the same redirector-handler configuration for the redirector loader *and* the VSAM Redirector Server. This means, you can:
1. Use a redirector loader to initially load a database.
2. Use a redirector handler to synchronize the database with the VSAM cluster.

These are the redirector loaders you can use:
- **RedirLoader**
- **MQLoader**
- **DeltaLoader**

Description of the RedirLoader Redirector Loader

The RedirLoader redirector loader provides a load functionality for redirector handlers. RedirLoader:
1. Uses the VSE Connector Client to read VSAM records in plain binary format.
2. Sends these records to a redirector handler, faking a VSAM INSERT request.

This provides a faster initial load of databases, whilst using the same redirector handlers that the Redirector uses. You do not need to configure the Redirector on z/VSE.

Description of the MQLoader Redirector Loader

The MQLoader redirector loader is used for inserting VSAM records that had been captured by the VSAM Capture Exit IESVSCAP:
1. MQLoader is called from the WebSphere MQ trigger-monitor program whenever new messages (records) are available in an MQ queue.
2. The VSAM Capture Exit IESVSCAP creates an MQ message for each VSAM INSERT, ERASE and UPDATE request.
3. The MQ messages are transferred by WebSphere MQ to another MQ Server running on a remote z/VSE host.
   a. If the request arrives at the remote z/VSE host, MQ triggers the MQLoader, which gets the request and feeds it into the configured redirector handler.
   b. If requests cannot be processed or contain errors, they will be stored in an error queue. These requests can then be later recovered.

Description of the DeltaLoader Redirector Loader

The DeltaLoader redirector loader is used for inserting VSAM records that had been captured by the VSAM Capture Exit IESVSCAP:

1. For each VSAM INSERT, ERASE or UPDATE request, the VSAM Capture Exit IESVSCAP creates a VSAM record in a separate delta VSAM cluster.
2. DeltaLoader reads the delta records from the VSAM delta cluster, and sends them to the configured redirector handler.
   a. If the delta records are processed successfully, they are deleted from the VSAM delta cluster.
   b. If requests cannot be processed or contain errors, they will be retained in the VSAM delta cluster. These requests can then be later recovered.

IBM-Supplied Example of DB2-Related Handler

This example (*DB2Handler*) shows how you can redirect all VSAM requests for a specific file, to a remote DB2 database. To use this example, you must have installed on the remote system:

- DB2 database
- Java Development Kit

DB2Handler is represented by the Java class **com.ibm.vse.db2handler.DB2Handler**, whose option string delivers:

- username
- tablename
- password
- systemname

To run *DB2Handler*, you must include the DB/2 JDBC 1.2 driver (**db2java.zip**) in your CLASSPATH variable.

DB2Handler starts various sub-handlers that handle ESDS/KSDS and RRDS/VRDS files. Each sub-handler then:

1. Opens a database table containing the field definitions.
2. Reads the database table.
3. Prepares the database table for record requests.
4. Analyzes each incoming request, and processes each request.
5. Sends the result back to the VSAM Redirector Client.

For further information, refer to the source code for each sub-handler, which contain detailed comments.

These are the restrictions for using *DB2Handler*:

- Share option 1 and 2 only are supported. Concurrent updates to the same DB2 table are not supported. However, your redirector handler can be coded to respond to database locks.
- STRING fields only are supported as KEY fields in KSDS files (for BASE Cluster and AIX).
- There is no support for NON-UNIQUE AIX access. You should avoid this type of access completely.
- Fields that contain 0x00 (`NULL` fields) are *not* supported as KEY fields (for BASE Cluster and AIX), because Key columns are defined as `NOT NULL`.

The *CreateDB2Tables* program (**com.ibm.vse.db2handler.create.CreateDB2Tables**) creates the field definition table on the DB/2 system. To use *CreateDB2Tables* you should:

1. Create an XML file containing the field definitions required to create your tables. A sample XML file is supplied with the DTD, which you can use as a template. You can define the field types STRING, UNSIGNED, SIGNED, PACKED and BINARY. In the database:
   - STRING is a CHAR
   - UNSIGNED, SIGNED and PACKED are INT
   - BINARY is mapped to a BLOB (binary large object).
2. Start program *CreateDB2Tables* and follow the instructions that this program provides.
3. Your tables (including field definitions and data table) will be created and, if required, indexes built.

Chapter 9. Installing the Database Call Level Interface

This chapter describes how you install the *Database Call Level Interface* (DBCLI). It allows z/VSE applications to access a relational database on any suitable database server. Therefore, you have the flexibility of being able to choose a database server (IBM DB2, Oracle, Microsoft SQL Server, MySQL, and so on) that runs on a platform other than z/VSE.

This chapter consists of these main topics:

- "Overview of the Database Call Level Interface"
- "Prerequisites for Using the Database Call Level Interface" on page 74
- "Installing the DBCLI Server" on page 75
- "Uninstalling the DBCLI Server" on page 76
- "Setting Up and Configuring the DBCLI Server" on page 76
- "DBCLI Server Commands" on page 81

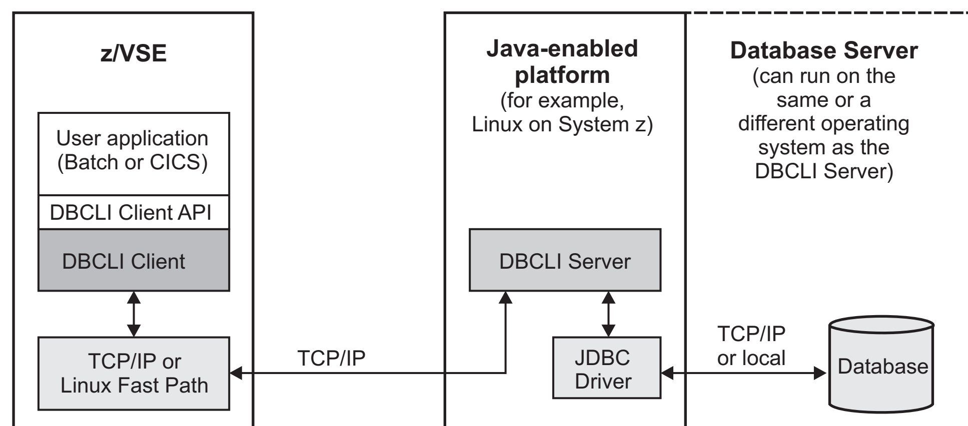
Related Topic:

- Chapter 22, "Using the Database Call Level Interface to Access Data," on page 291

Overview of the Database Call Level Interface

The z/VSE Database Call Level Interface (*DBCLI*) consist of two main parts:
- The *DBCLI client* that runs under z/VSE.
- The *DBCLI server* (program `DBCliServer`) that runs on a Java platform.

The DBCLI client provides a programming API for your application programs:
1. The DBCLI client connects to the DBCLI server via a TCP/IP connection.
2. The DBCLI client translates the calls from your application programs into requests to the DBCLI server.
3. The DBCLI server receives the requests from the applications and passes them to a JDBC driver that is provided by the Vendor database.
4. The DBCLI server returns the result of the call to the application program running under z/VSE via the DBCLI client.



© Copyright IBM Corp. 2000, 2013

Because neither the DBCLI client nor the DBCLI server inspect the SQL statements being used by your application program, you can use *any type* of SQL statements and SQL dialect that are supported by a Vendor's JDBC Driver (IBM DB2, Oracle, Microsoft SQL Server, MySQL, and so on) *providing* the database-provider also supplies a suitable JDBC driver.

Overview of Connection Pooling

Connection pooling keeps and reuses *existing* database connections for DBCLI applications running under the CICS Transaction Server for VSE/ESA V1.1. This can be especially beneficial for short-lived, CICS DBCLI applications that run frequently.

Without connection pooling, the creation of database connections can be comparatively time-consuming because of the overhead required to establish a network connection and initialize access to a database server.

The use of the connection pool is transparent to the DBCLI application. The default is to not use connection pooling, therefore existing DBCLI applications will continue to work unchanged (that is, without using the connection pool).

The connection pool is only available under the CICS. Batch applications *cannot* make use of the connection pool and any attempt to use the connection pool will be rejected.

You can implement connection pooling from *z/VSE 5.1 onwards*. However, if you wish to implement connection pooling in a z/VSE 5.1 system, you must install the June 2013 z/VSE 5.1 Service Upgrade.

Connection pooling is implemented by configuring the *connection pool manager*.

Related Topics:

- “Configuring and Starting/Stopping the Connection Pool Manager” on page 78.
- “Connection Pooling” on page 295 (programming concepts).

Prerequisites for Using the Database Call Level Interface

These are the prerequisites for using the Database Call Level Interface (DBCLI):

- You must supply your own Vendor database that is to be used with the Database Call Level Interface.
- The Vendor database (IBM DB2, Oracle, Microsoft SQL Server, MySQL, and so on) must provide a JDBC driver that *supports JDBC V3.0 or later*.
- The DBCLI server requires a Java-enabled platform that has a Java Runtime (JRE) or Java Developer Kit (JDK) of version 1.5 or later.

If you wish to implement *connection pooling* with the DBCLI, you must configure and start the DBCLI *connection pool manager*. For details, see “Configuring and Starting/Stopping the Connection Pool Manager” on page 78.

Installing the DBCLI Server

This topic describes how you install the DBCLI server on a physical/logical tier of a 2-tier or 3-tier environment.

The DBCLI server is included in the VSE Central Functions and consists of one file **IESDBSRV.w**.

Obtaining a Copy of the DBCLI Server

To obtain a copy of the DBCLI server, you must decide if you wish to obtain it:
- from the Internet.
- by installing the VSE Connectors Workstation Code from the Extended Base Tape.

To obtain the DBCLI server *from the Internet*, you should:
1. Start your Web browser and go to the URL

   `http://www.ibm.com/systems/z/os/zvse/downloads/`
2. From within the **Database Connector** section, download the file **DatabaseCliServer*nnn*.zip** to the directory where you wish to install the DBCLI server. **Note:** *nnn* refers to the current VSE version (for example, **DatabaseCliServer511.zip**).

To obtain the DBCLI server *by installing the VSE Connectors Workstation Code component*, you should:
1. Install the VSE Connectors Workstation Code component from the Extended Base Tape. After you have installed this component, the DBCLI server W-book **IESDBSRV.w** will be stored in z/VSE sublibrary PRD2.PROD.
2. Use the FTP (file transfer program) utility of TCP/IP for VSE/ESA to download **IESDBSRV.w** to the directory where you wish to install the DBCLI server. Then rename **IESDBSRV.w** to **DatabaseCliServer.zip**.

Notes:
1. You must download **IESDBSRV.w** in *binary*.
2. Make sure that Unix mode is ***turned off***. Otherwise **IESDBSRV.w** will be downloaded in ASCII mode, even when you specify *binary*. *Unix mode* is one parameter of your VSE FTP daemon. Some FTP clients might *force* Unix mode to be turned on! The example below shows how a successful transfer of **IESDBSRV.w** was made using a batch FTP client. The place where the UNIX mode is set, is shown as bold.

```
c:\temp>ftp 9.164.155.2
Connected to 9.164.155.2.
220-TCP/IP for VSE -- Version 01.05.F  -- FTP Daemon
    Copyright (c) 1995,2006 Connectivity Systems Incorporated
220 Service ready for new user.
User (9.164.155.2:(none)): sysa
331 User name okay, need password.
Password:
230 User logged in, proceed.
ftp> cd prd2
250 Requested file action okay, completed.
ftp> cd prod
250 Requested file action okay, completed.
ftp> binary
200 Command okay.
ftp> get iesdbsrv.w
200 Command okay.
```

```
150-File: PRD2.PROD.IESDBSRV.W
    Type: Binary Recfm: FB Lrecl:    80 Blksize:     80
    CC=ON  UNIX=OFF RECLF=OFF TRCC=OFF CRLF=ON  NAT=NO
150 File status okay; about to open data connection
226-Bytes sent:       n,nnn,nnn
    Records sent:        nn,nnn
    Transfer Seconds:        nn.nn (    nnnK/Sec)
    File I/O Seconds:         n.nn (  n,nnnK/Sec)
226 Closing data connection.
nnnnnnn bytes received in nn,nn seconds (nnn,nn Kbytes/sec)
ftp> bye
221 Service closing control connection.
c:\temp>ren IESDBSRV.w DatabaseCliServer.zip
```

Performing the DBCLI Server Installation

To perform the installation of the DBCLI server, you must:

1. Unzip the file **DatabaseCliServer.zip**, which contains these files:
   - setup.jar (contains the DBCLI server code)
   - setup.bat (an install batch file for Windows XP/Vista/7)
   - setup.cmd (an install batch file for Windows NT)
   - setup.sh (an install script for Linux/Unix)
2. Start the batch file (by double-clicking the file) that is applicable to your operating-system platform.
3. The installation process now begins, and you are guided through various installation menus.

Uninstalling the DBCLI Server

You have two possible methods to uninstall the DBCLI server:

- By using the normal *Windows* **Add or Remove Programs** function, which is selected via the Windows Control Panel.
- By manually running the DBCLI server uninstall program. You can find this program in the **_uninst** subdirectory that is located within the directory where you installed the DBCLI server. *This option is available for all platforms*.

Setting Up and Configuring the DBCLI Server

The DBCLI server is installed on a Java-enabled platform. The installation is done by executing a Java-based installer. This installer is contained in the file setup.jar and can be invoked by running one of the install scripts:

- setup.bat (Windows)
- setup.cmd (OS/2 and Windows)
- setup.sh (Linux, Unix).

When running the Java installer, various dialog boxes are displayed to guide you through the installation.

You configure the DBCLI server using these files:
- DatabaseCliServer.cfg
- JdbcAliases.cfg
- JdbcDriver.cfg

These files are now described in detail.

DatabaseCliServer.cfg

The DatabaseCliServer.cfg configuration file contains basic settings such as the port number where the server listens, default code page, and so on.

```
######################################################################
# VSE Database CLI Server configuration
######################################################################

# Port where the Database CLI Server listens
listenport=16178

# Number of maximum parallel connections allowed.
maxconnections=256

# Optional. Address to bind the listening socket to.
#bindaddr=127.0.0.1

# Optional. Default EBCDIC codepage used with the VSE application.
#ebcdiccodepage=Cp1047

# Print messages (on) or do not print messages (off).
messages=on

# Print trace (on) or do not print trace (off).
trace=off

# Sets the trace level which can be:
# FLOW - trace the main application flow
# NORMAL - most messages
# FINE - most extensive tracing
tracelevel=NORMAL

# Optional. Configuration file for the JDBC alias definitions.
#jdbcaliases=JdbcAliases.cfg

# Configuration file for the JDBC drivers.
#jdbcdriver=JdbcDriver.cfg

######################################################################
#SSL specific settings (uncomment to activate):
######################################################################

# sslversion can be either SSL or TLS.
#sslversion=SSL

# If client authentication is true, the server requests the client to send its
certificate and verifies it.
# If client authentication is false or not specified, no client authentication
is done.
#clientauthentication=true

# keyringfile specifies the file name of the keyring file.
#keyringfile=keyring.pfx

# keyringpwd specifies the password for opening the keyring file.
#keyringpwd=ssltest

# ciphersuites specifies a comma separated list of cipher suites that are
accepted by the server.
# If this property is not specified, all supported cipher suites are used.
#ciphersuites=SSL_RSA_WITH_NULL_MD5,SSL_RSA_WITH_NULL_SHA,SSL_RSA_EXPORT_WITH_DES40_CBC_SHA,
SSL_RSA_WITH_DES_CBC_SHA,SSL_RSA_WITH_3DES_EDE_CBC_SHA,TLS_RSA_WITH_AES_128_CBC_SHA

#-------------------------------------------------------------------
# Available cipher suites with VSE (other systems may support
# different/additional cipher suites:
#
# Suite                              Handshaking    Hash  Compression
# ---------------------------------- -------------- ----- -----------
# SSL_RSA_WITH_NULL_MD5              512-bit        MD5   No
# SSL_RSA_WITH_NULL_SHA              512-bit        SHA   No
# SSL_RSA_EXPORT_WITH_DES40_CBC_SHA  512-bit        SHA   40-bit
# SSL_RSA_WITH_DES_CBC_SHA           1024-bit       SHA   56-bit
# SSL_RSA_WITH_3DES_EDE_CBC_SHA      1024-bit       SHA   68-bit
# TLS_RSA_WITH_AES_128_CBC_SHA(*)    1024/2048-bit  SHA   128-bit
#
# (*) this cipher suite is new with TCP/IP for VSE service pack 1.5E
```

```
# and requires the CPACF feature, which is available on zSeries
# processors z890, z990, and System z9. For 2048-bit SSL
# handshaking a cryptographic accelerator card (PCICA or Crypto
# Express2 is required.
```

JdbcAliases.cfg

The JdbcAliases.cfg configuration file contains alias definitions for the databases you want to connect to. Every alias points to a database by using the JDBC URL as defined by the JDBC driver. You must specify the alias name at the CONNECT function of the DBCLI programming interface when the application program on z/VSE attempts to establish a connection to a database. You can optionally define a user name and password for each alias. If you do so, then the application program running on z/VSE does not require to specify a user name and password at the CONNECT function.

The DBCliServer can dynamically reload the JdbcAliases.cfg configuration file. Use the 'reload jdbcalias' command to reload the file.

```
SAMPLE.jdbcurl = jdbc:db2:SAMPLE


SAMPLE2.jdbcurl = jdbc:db2://ifranzki/SAMPLE
SAMPLE2.user = HUGO
SAMPLE2.password = PASSWORD
```

JdbcDriver.cfg

The JdbcDriver.cfg configuration file contains the class names of all JDBC drivers that should be loaded by the DBCliServer. It also contains hints about the format of the JDBC url to use when you want to connect to a database using these JDBC drivers.

You must make sure that the JDBC Driver's class or JAR files are contained in the CLASSPATH environment variable. If a JDBC driver could not be found, you will see an appropriate message during startup of the DBCliServer.

Configuring and Starting/Stopping the Connection Pool Manager

This topic describes how you configure, start/stop, and query the *connection pool manager*.

The (DBCLI) connection pool manager is a long-running transaction that manages the connection pool.

- There is one connection pool manager per CICS region.
- The connection pool manager can “pool” connections to different DBCLI servers and databases.

Related Topics:

- “Overview of Connection Pooling” on page 74.
- “Connection Pooling” on page 295 (programming concepts).

Defining CICS Programs/Transactions and OME for Connection Pooling

To implement DBCLI connection pooling via the connection pool manager, you must define various programs and transactions to CICS.

These are the programs that must be defined to CICS:

```
IESDBCPM Language=Assembler, EXECKey=CICS, DataLoc=Any
IESDBCPP Language=Assembler, EXECKey=User, DataLoc=Any
IESDBCPQ Language=Assembler, EXECKey=User, DataLoc=Any
IESDBCPS Language=Assembler, EXECKey=User, DataLoc=Any
```

The following transactions must be defined both to CICS and to the Basic Security Manager (BSM):

```
IDBM Program=IESDBCPM, TaskDataKey=User, TaskdataLoc=ANY
IDBP Program=IESDBCPP, TaskDataKey=User, TaskdataLoc=ANY
IDBQ Program=IESDBCPQ, TaskDataKey=User, TaskdataLoc=ANY
IDBS Program=IESDBCPS, TaskDataKey=User, TaskdataLoc=ANY
```

To obtain DBCLIDEF.JOB so that you can perform the definitions shown above, you must:

1. Start your Web browser and go to the URL

   `http://www.ibm.com/systems/z/os/zvse/downloads/`
2. From within the **Database Connector** section, download **DBCLIDEF.JOB** and execute on your z/VSE 5.1 system.

If you wish to install the online messages (OME) required for connection pooling, you must:

1. Start your Web browser and go to the URL

   `http://www.ibm.com/systems/z/os/zvse/downloads/`
2. From within the **Database Connector** section, download **DBCLIOME.BJB** and execute on your z/VSE 5.1 system.

Starting the Connection Pool Manager

You can start the DBCLI connection pool manager either manually using transaction IDBS, or automatically during CICS startup. If you start the DBCLI connection pool manager automatically, you can start EZATRUE program at the same time.

To *manually start* the connection pool manager, you use transaction **IDBS** . This transaction can be invoked from the system console or any other terminal.

Note: You must have started the EZA Task Related User Exit (EZATRUE) *before* you start the connection pool manager! For details of how to start EZATRUE, refer to “CICS Considerations for the EZA Interfaces” in the *z/VSE TCP/IP Support*.

```
►►──IDBS──┬─────────┬──┬───┬──┬─────────┬──┬───┬──┬───────┬──┬───┬──┬────────┬──►◄
          └─TCPNAME─┘  └─,─┘  └─ADSNAME─┘  └─,─┘  └─FORCE─┘  └─,─┘  ├─SYSLOG─┤
                                                                    ├─SYSLST─┤
                                                                    └─BOTH───┘
```

Where:

TCPNAME
: Specifies the local TCP/IP stack that is used with the DBCLI applications.

This parameter can be set to `SOCKETnn` or simply to *nn* (left- or right-justified, and padded with six blanks). The value *nn* determines the ID of the selected TCP/IP stack as it is specified with the ID parameter in the TCP/IP startup job. If TCPNAME is not specified, the ID from the `// OPTION SYSPARM='nn'` statement is used. The ID defaults to `'00'` if none of the above are specified.

ADSNAME
Specifies the name of the TCP/IP Interface Routine used by the EZA processing environment. Per default, the IBM-supplied TCP/IP Interface Routine EZASOH99 is used. You can overwrite this specification using the statement `// SETPARM EZA$PHA='routine-name'`.

FORCE
Specifies that the connection manager pool should be force-started, when the pool manager has been previously canceled.

Note: This start option should be used *with caution* and only if the connection pool manager is no longer responding. This start option might destroy an active connection pool manager!

SYSLOG | SYSLST | BOTH
Specifies that tracing of the connection pool manager should be activated. For SYSLOG, trace messages will be written to the system console. For SYSLST, trace messages will be written to the CICS listing. For BOTH, trace messages will be written to the system console and to the CICS listing.

To *automatically start* EZATRUE and the connection pool manager during CICS startup, add programs EZASTRUE and IESDBCPS to table DFHPLTPI using these statements:

```
DFHPLT TYPE=ENTRY,PROGRAM=EZASTRUE
DFHPLT TYPE=ENTRY,PROGRAM=IESDBCPS
```

Note: If you decide to automatically start the connection pool manager and ESATRUE, you *cannot* specify any of the startup parameters described previously.

Stopping the Connection Pool Manager

You can stop the DBCLI connection pool manager either manually using transaction IDBP, or automatically during CICS shutdown.

To *manually stop* the connection pool manager, you use transaction **IDBP**. This transaction can be invoked from the system console or any other terminal. Any currently-pooled connections will be closed. If you wish, you can also manually stop EZATRUE - for details, refer to “CICS Considerations for the EZA Interfaces” in the *z/VSE TCP/IP Support*.

To *automatically stop* the connection pool manager and EZATRUE during CICS shutdown, add programs IESDBCPP and EZASTRUE to table DFHPLTSD using these statements:

```
DFHPLT TYPE=ENTRY,PROGRAM=IESDBCPP
DFHPLT TYPE=ENTRY,PROGRAM=EZASTRUE
```

Querying the Connection Pool

To query the DBCLI connection pool, you use transaction **IDBQ**. This transaction can be invoked from the system console or any other terminal.

If transaction IDBQ is used from the system console, it will display a list of currently-pooled connections on the system console.

If transaction IDBQ is used from a terminal, it will display a list of currently-pooled connections in a 3270 screen. You can then scroll through this list.

DBCLI Server Commands

When the DBCLI server (`DBCliServer`) is up and running, you can enter the following commands to operate the server:

- **status** displays the status of all connections
- **stop all** disconnects all active connections
- **stop <n>** disconnects the specified connection
- **quit [force]** quits the server
- **trace <on|off>** turns trace on or off
- **show jdbcalias** displays the JDBC alias configuration
- **update jdbcalias** reloads the JDBC alias configuration
- **help** displays help information

Chapter 10. Customizing the DB2-Based Connector

This chapter describes the *customization* activities required before the DB2-based connector can be used.

It contains these main topics:

- "Overview of the DB2-Based Connector"
- "Host Installation Activities That Must Be Already Completed" on page 84
- "Step 1: Customize CICS TS" on page 84
- "Step 2: Customize TCP/IP" on page 85
- "Step 3: Customize DB2 and Define Sample Database" on page 85
- "Step 4: Set Up for DRDA Support" on page 93
- "Step 5: Set Up Stored Procedure Server and Define to DB2" on page 93
- "Step 6: Set Up for Stored Procedures" on page 94
- "Step 7: Customize the DB2-Based Connector for VSAM Data Access" on page 95
- "Step 8: Customize the DB2-Based Connector for DL/I Data Access" on page 95
- "Step 9: Start DB2, and Start Stored Procedure Server" on page 96
- "Step 10: Install DB2 Connect and Establish Client-Host Connection" on page 96

Overview of the DB2-Based Connector

The *DB2-based connector* is an optional feature that allows you to use the *Distributed Relational Database Architecture* (DRDA®) to access non-relational data such as VSE/VSAM and DL/I data. Your application programs use standard interfaces such as JDBC, ODBC, or Call Level Interface (CLI) to request data.

The implementation of the DB2-based connector is based upon the use of *DB2 Stored Procedures*, which you can use with the DB2 Server for VSE & VM, Version 6 or later. DB2 Stored Procedures are application programs that you write, and then compile and store on your z/VSE host.

The DB2-based connector requires that the *DB2 Server for VSE* runs on z/VSE.

Note: The DB2 Server for VSE Client Edition is **not** sufficient for use with the DB2-based connector, since the DB2 Server for VSE Client Edition does **not** allow DB2 Stored Procedures to be run on z/VSE.

You can write DB2 Stored Procedures in any LE (Language Environment®)-compliant language (COBOL, C, or PL/I). Local or remote DRDA applications can then invoke these DB2 Stored Procedures.

The DB2-based connector enables you to access VSE/VSAM and DL/I data from within the *same* DB2 Stored Procedure that you use to access DB2 data:

- To access VSE/VSAM data, you use the *VSAM Call Level Interface*. See "Using DB2 Stored Procedures to Access VSAM Data" on page 395 for details.

© Copyright IBM Corp. 2000, 2013

- To access DL/I data, you use the *AIBTDLI interface*. See "Using DB2 Stored Procedures to Access DL/I Data" on page 402 for details.

For an overview of where the DB2-based connector is used in a 3-tier environment, see Figure 3 on page 9.

Host Installation Activities That Must Be Already Completed

This topic provides a summary of the installation activities that must already be completed on the z/VSE host, before you begin to customize the DB2-based connector:

- The DB2 Server for VSE must be restored from the z/VSE Extended Base Tape to sub-library PRD2.DB2750. The startup job for the DB2 Server for VSE is defined for dynamic partition of class S. It is either installed:
  - During the initial installation of z/VSE
  - Following an FSU (Fast Service Upgrade).
- The DB2-based connector requires that the *DB2 Server for VSE* runs on z/VSE.

  Note: The DB2 Server for VSE Client Edition is **not** sufficient for use with the DB2-based connector, since the DB2 Server for VSE Client Edition does **not** allow DB2 Stored Procedures to be run on z/VSE.
- The AIBTDLI interface must be installed for accessing DL/I data via DB2 Stored Procedures. To use the AIBTDLI interface:
  - DL/I VSE must be installed.
  - Your CICS/DLI system must have all databases (DBDs) that you wish to use defined in the CICS FCT, together with the AIBTDLI interface.
  - Your CICS/DLI system must have:
    - all PSBs defined in the DL/I online nucleus DLZNUCxx
    - an active MPS system.
  - The DL/I task termination exit *DLZBSEOT* (described in "Task Termination and Abend Handling" on page 410) must be resident in the SVA.
- One or more CICS TS systems must be customized for use with the DB2 Server for VSE.

Step 1: Customize CICS TS

If you have more than one CICS TS running, you must first decide which (one or more) of your CICS systems is to have access to the *DB2 Server for VSE*. To customize a selected CICS TS you must:

- Have journaling *active*. Use the corresponding DFHSITxx skeleton to set JCT=**SP** (for DFHJCT**SP**) or another suffix.
- Compile the JCT using the modified skeleton DFHJCTSP.
- Define the journal files. For CICSICCF use the provided skeleton SKJOURN. For PRODCICS use the provided skeleton SKJOUR2. Both are available in VSE/ICCF library 59.

Step 2: Customize TCP/IP

Ensure that your TCP/IP startup job has the following two values set:

```
SET WINDOW           = 8192
SET MAX_SEGMENT      = 700
```

For editing, use the TCP/IP dialog on your client workstation, or modify the appropriate startup member in your VSE library.

For details, refer to the *z/VSE TCP/IP Support*.

Step 3: Customize DB2 and Define Sample Database

Step 3 involves running **SKDB2VAR** (*in partition BG*), which is the main skeleton you use for customizing the DB2-based connector. The jobs included in this skeleton are now described.

Notes:

1. Before starting this step, you must have activated the license key for using DB2. If not, refer to the *z/VSE Planning* for details of how to activate this key using the skeleton SKUSERBG (contained in ICCF Library 59).
2. Skeleton SKDB2VAR assumes you are using an IBM 3380 disk device for storage allocations. If you use a different disk device type, you must change these allocations accordingly (especially in the case of FBA devices).

These are the jobs that skeleton SKDB2VAR runs:

- "Step 3.1: Define User Catalog"
- "Step 3.2: Catalog New ARISIVAR.Z" on page 86
- "Step 3.3: Job Manager for Preparation / Installation Steps" on page 87
- "Step 3.4: Activate DRDA Server Support" on page 87
- "Step 3.5: Startup Job for Stored Procedure Server" on page 87
- "Step 3.6: Prepare DB2 Sample Database" on page 88
- "Step 3.7: Install DB2 Sample Database" on page 89

Step 3.1: Define User Catalog

The job **DB2DEFCT** defines a User Catalog and space on a separate volume.

For the DB2UCAT catalog, the allocated space consists of 150 cylinders.

A standard label is inserted for the new catalog, with the name **DB2UCAT**. You must enter your own values for these variable:

- -V001- -
: The Id of your volume. This VOLID is also used in job DB2CTVAR.

- -V002- -
: The number of tracks allocated for the catalog space. The allocation is 150 cylinders.

```
$$ JOB JNM=DB2DEFCT,CLASS=0,DISP=D,NTFY=YES
$$ LST CLASS=Q,DISP=H
// JOB DB2DEFCT DEFINE USER CATALOG DB2UCAT
* THIS JOB WILL TERMINATE IN CASE THE DB2UCAT IS ALREADY DEFINED.
// EXEC IDCAMS,SIZE=AUTO
  LISTCAT ALL CATALOG(DB2.USER.CATALOG)
  IF LASTCC EQ 8 THEN DO
```

```
      SET LASTCC = 0
      SET MAXCC = 0
   DEFINE USERCATALOG(NAME(DB2.USER.CATALOG) -
      VOL(--V001--)                          -
      NOTRECOVERABLE                         -
      TRK(15))
   DEFINE SPACE(VOLUMES(--V001--)            -
      CYL(--V002--))                         -
      CATALOG(DB2.USER.CATALOG)
    END
   ELSE DO
      SET LASTCC = 0
      SET MAXCC = 0
   END
/*
// OPTION STDLABEL=DELETE
DB2UCAT
/*
// OPTION STDLABEL=ADD
// DLBL DB2UCAT,'DB2.USER.CATALOG',0,VSAM
/*
// EXEC IESVCLUP,SIZE=AUTO             ADD LABEL TO STDLABUP PROC
D                                              DB2UCAT
D DB2.SQLGLOB.MASTER                           SQLGLOB
A DB2.USER.CATALOG                             DB2UCAT
A DB2.SQLGLOB.MASTER                           SQLGLOB DB2UCAT OLD KEEP
/*
/&
$$ EOJ
```

Step 3.2: Catalog New ARISIVAR.Z

In this step, Job DB2CTVAR first renames the original DB2-supplied ARISIVAR.Z to ARISIVAR.ORIG, and catalogs the global variable member ARISIVAR.Z. Then the new ARISIVAR.Z is used to test the installation of the DB2-based connector and the sample database.

Note that the processing of ARISIVAR.Z is controlled by the **DB2 Job Manager**.

You start the Job Manager by releasing the job **DB2JMGR** in the VSE/POWER reader queue (placed there by skeleton SKDB2VAR). You must release the Job Manager once for each step: Preparation, Installation (or Migration).

As described in detail in the *Program Directory* for the *DB2 Server for VSE*, ARISIVAR.Z processes many parameters, globals and variables, to define DB2 characteristics and resources.

Major definitions include, for example:

- The DB2 sample database **SQLDS** which is defined on volume **- -V001- -** (variable in skeleton SKDB2VAR).
- The **DB2 Server for VSE Help** component, the installation of which is controlled by the following variable:

  ```
  ARIS75JZ HELP       YES
  ```

  You are recommended to install the DB2 Server for VSE Help (you must replace **- -V003- -** with the address of your tape drive). When this variable is processed, you are requested to mount the corresponding tape (containing the DB2 Help files). This is the third tape (the extra tape) of the Base distribution tapes you received.

- The creation of DB2 (work) files such as **BINDFILE, BINDWFILE**, and **SQLGLOB**.
- The setting of **CICS TS** parameters as required for a DB2 environment.

Here are the contents of DB2CTVAR:

```
$$ JOB JNM=DB2CTVAR,CLASS=0,DISP=D,NTFY=YES
$$ LST CLASS=Q,DISP=H
// JOB DB2CTVAR CATALOG GLOBAL VARIABLE MEMBER FOR DB2
* ORIGINAL ARISIVAR.Z RENAMED TO ARISIVAR.ORIG
// EXEC  LIBR,PARM='MSHP'
    ACCESS  SUBLIB = PRD2.DB2750
    RENAME          ARISIVAR.Z:=.ORIG
CATALOG ARISIVAR.Z       EOD=&&         REPLACE=YES
⋮


 (for further details, refer to the sample SKDB2VAR in Library 59)
```

Step 3.3: Job Manager for Preparation / Installation Steps

```
$$ JOB JNM=DB2JMGR,CLASS=R,DISP=L,NTFY=YES
$$ LST CLASS=Q,DISP=H
// JOB DB2JMGR DB2 JOB MANAGER
// LIBDEF *,SEARCH=(PRD2.DB2750)
// EXEC REXX=ARISIMGR
/*
/&
$$ EOJ
```

Job DB2JMGR is used in “Step 3.6: Prepare DB2 Sample Database” on page 88 and “Step 3.7: Install DB2 Sample Database” on page 89.

Step 3.4: Activate DRDA Server Support

This job will be later used in Step 4.

```
$$ JOB JNM=DB2DRDA,CLASS=R,DISP=L,NTFY=YES
$$ LST CLASS=Q,DISP=H
// JOB DB2DRDA ACTIVATE DRDA SERVER SUPPORT
* **********************************************************
*  LINK EDIT RDS WITH DRDA SERVER SUPPORT
* **********************************************************
// LIBDEF *,SEARCH=PRD2.DB2750
// LIBDEF PHASE,CATALOG=PRD2.DB2750
// OPTION CATAL
 INCLUDE ARISLKRA
// EXEC  PGM=LNKEDT,PARM='MSHP,AMODE=31,RMODE=ANY'
/*
/&
$$ EOJ
```

Step 3.5: Startup Job for Stored Procedure Server

In this step, skeleton SKDB2VAR loads the job to start the Stored Procedure Server, into the VSE/POWER reader queue.

```
$$ JOB JNM=PSERVER,CLASS=0,DISP=L
$$ LST CLASS=W,DISP=H
// JOB PSERVER
// LIBDEF PROC,SEARCH=(PRD2.DB2750)
// DLBL SQLGLOB,'DB2.SQLGLOB.MASTER',,VSAM,CAT=DB2UCAT,DISP=(OLD,KEEP)
// EXEC PROC=ARIS75PL       *-- DB2 PROD. LIBRARY ID PROC
// EXEC PROC=ARIS75DB       *-- DB2 DATABASE ID PROC
// EXEC  ARIDBS,SIZE=AUTO
CONNECT SQLDBA IDENTIFIED BY SQLDBAPW;
```

```
CREATE PSERVER SPSERV01 AUTOSTART YES;
/*
/&
$$ EOJ
```

Step 3.6: Prepare DB2 Sample Database

This job uses the preparation-related global definitions contained within ARISIVAR.Z (catalogued in Step 3.2). To run the preparation step, DB2 Job Manager (DB2JMGR) must be released. Refer to the console listing below for details of how to run this step.

Note: You must enter a **0** (partition **BG**) to run the *preparation* job – in the sample below, the DB2 Job Manager runs in partition F4, and DB2 Server for VSE Version 7.5 is used.

```
r rdr,db2jmgr
AR 0015 1C39I COMMAND PASSED TO VSE/POWER
F1 0001 1R88I  OK
F4 0001 1Q47I F4 DB2JMGR 00249 FROM (HEHA) , TIME= 9:28:39, TKN=000001A7
F4 0004 // JOB DB2JMGR DB2 JOB MANAGER
        DATE 07/07/2001, CLOCK 09/28/39
F4 0004 **************************************************************
F4 0004       PREPARE FOR INSTALLATION/MIGRATION PROCESS
F4 0004 **************************************************************
F4 0004 ENTER INSTALLATION LIBRARY NAME (PRD2.DB2750   default)
F4-0004
4
F4 0004 YOU HAVE SELECTED  PRD2.DB2750
F4 0004 PRESS ENTER TO CONTINUE OR ENTER ANY OTHER KEY TO
F4 0004 MODIFY YOUR SELECTION:
F4-0004
4
F4 0004 WHICH CLASS WILL YOU USE TO RUN THE PROCESS ?  (4  default)
F4-0004
4 0
F4 0004 YOU HAVE SELECTED  0
F4 0004 PRESS ENTER TO CONTINUE OR  ENTER ANY OTHER KEY TO
F4 0004 MODIFY YOUR SELECTION:
F4-0004
4
F4 0004 PLEASE SELECT ONE OF THE FOLLOWING :
F4 0004 FOR PREPARATION....  ENTER (P)
F4 0004 FOR INSTALLATION...  ENTER (I)
F4 0004 FOR MIGRATION......  ENTER (M)
F4-0004
F4-0004
4 p
F4 0004 IF PREPARATION FOR:
F4 0004 INSTALLATION... PLEASE ENTER (I)
F4 0004 MIGRATION...... PLEASE ENTER (M)
F4-0004
4 i
F4 0004 DO YOU WANT TO EXECUTE ALL JOBS? {Y|N-default}
F4-0004
4 y
F4 0004
F4 0004 ************ JOB ARIS75JD ********************
F4 0004 *   DEFINE DB2750 PROGRAMS AND TRANSACTIONS
F4 0004 **********************************************
F4 0004 JOB ARIS75JD IS OPTIONAL.
F4 0004 WOULD YOU LIKE TO RUN IT? {Y|N-Default}
F4-0004
4 y
BG 0001 1Q47I   BG ARIS75JD 00281 FROM (HEHA) , TIME=11:12:37, TKN=000001A8
BG 0000 * ** JOB JNM=ARIS75JD,CLASS=0,DISP=D
BG 0000 * ** LST CLASS=V,DISP=D,DEST=(,XXXXXXX)
:
F4 0004 JOB ARIS759D IS OPTIONAL.
```

```
F4 0004 WOULD YOU LIKE TO RUN IT? {Y|N-Default}
F4-0004
4 y
BG 0001 1Q47I   BG ARIS759D 00285 FROM (HEHA) , TIME=11:13:10, TKN=000001A9
BG 0000 // JOB ARIS759D
BG 0000 * **************************************************************
BG 0000 * ARIS759D: DEFINE VSAM CLUSTER FOR THE BINDWKF FILE
BG 0000 * **************************************************************
BG 0000 EOJ ARIS759D  MAX.RETURN CODE=0000
BG 0000 EOJ NO NAME
BG 0001 1Q34I   BG WAITING FOR WORK
F4 0004 ***********************************************
F4 0004 * Job  ARIS759D  executed successfully
F4 0004 ***********************************************
F4 0004
F4 0004 ************ JOB ARISIQBD ********************
F4 0004 *   ISQL BIND FILE CONVERSION
F4 0004 ***********************************************
F4 0004 JOB ARISIQBD IS OPTIONAL.
F4 0004 WOULD YOU LIKE TO RUN IT? {Y|N-Default}
F4-0004
4 y
BG 0001 1Q47I   BG ARISIQBD 00286 FROM (HEHA) , TIME=11:13:20, TKN=000001AA
BG 0000 // JOB ARISIQBD -- ISQL BIND FILE CONVERSION
BG-0000 // PAUSE
0
BG 0000 EOJ ARISIQBD  MAX.RETURN CODE=0000
BG 0000 EOJ NO NAME
BG 0001 1Q34I   BG WAITING FOR WORK
F4 0004 ***********************************************
F4 0004 * Job  ARISIQBD  executed successfully
F4 0004 ***********************************************
F4 0004
F4 0004 ************ JOB ARIS75CD ********************
F4 0004 *   DB2 SERVER STARTER DATABASE VSAM DEFINITIONS
F4 0004 ***********************************************
BG 0001 1Q47I   BG ARIS75CD 00294 FROM (HEHA) , TIME=11:28:08, TKN=000001AB
BG 0000 // JOB ARIS75CD             DB2 FOR VSE STARTER DB VSAM DEFINITIONS
BG 0000 EOJ ARIS75CD  MAX.RETURN CODE=0
F4 0004
F4 0004 ************ JOB ARISSTDL ********************
F4 0004 *   ADD NEW LABELS TO STANDARD LABELS
F4 0004 ***********************************************
F4 0004 JOB ARISSTDL IS OPTIONAL.
F4 0004 WOULD YOU LIKE TO RUN IT? {Y|N-Default}
F4-0004
4 y
BG 0001 1Q47I   BG ARISSTDL 00297 FROM (HEHA) , TIME=11:30:06, TKN=000001AC
BG 0000 // JOB ARISSTDL
BG 0000 EOJ ARISSTDL
BG 0000 EOJ NO NAME
BG 0001 1Q34I   BG WAITING FOR WORK
F4 0004 READ CONSOLE FOR DETAILS.
F4 0004 EOJ DB2JMGR   MAX.RETURN CODE=0000
```

Step 3.7: Install DB2 Sample Database

This job uses the installation-related global definitions contained within ARISIVAR.Z (catalogued in Step 3.2). To run the installation step, DB2 Job Manager (DB2JMGR) must be released. Refer to the console listing below for details of how to run this step.

Note: You must run this installation step in a *static* partition (the example below uses static partition **F4**) – in the sample below, the DB2 Job Manager runs in partition F7.

```
r rdr,db2jmgr
AR 0015 1C39I COMMAND PASSED TO VSE/POWER
F1 0001 1R88I  OK
```

```
F7 0007 // JOB DB2JMGR DB2 JOB MANAGER
F7 0007 **************************************************************
F7 0007       PREPARE FOR INSTALLATION/MIGRATION PROCESS
F7 0007 **************************************************************
F7 0007 ENTER INSTALLATION LIBRARY NAME (PRD2.DB2750   default)
F7-0007
7
F7 0007 YOU HAVE SELECTED  PRD2.DB2750
F7 0007 PRESS ENTER TO CONTINUE OR ENTER ANY OTHER KEY TO
F7 0007 MODIFY YOUR SELECTION:
F7-0007
7
F7 0007 WHICH CLASS WILL YOU USE TO RUN THE PROCESS ?  (4  default)
F7-0007
7 4
F7 0007 YOU HAVE SELECTED  4
F7 0007 PRESS ENTER TO CONTINUE OR  ENTER ANY OTHER KEY TO
F7 0007 MODIFY YOUR SELECTION:
F7-0007
7
F7 0007 PLEASE SELECT ONE OF THE FOLLOWING :
F7 0007 FOR PREPARATION....  ENTER (P)
F7 0007 FOR INSTALLATION...  ENTER (I)
F7 0007 FOR MIGRATION......  ENTER (M)
F7-0007
7 i
F7 0007 DO YOU WANT TO EXECUTE ALL JOBS? {Y|N-default}
F7-0007
7 y
F7 0007
F7 0007 ************ JOB ARISBDID ********************
F7 0007 * SETUP THE DBNAME DIRECTORY
F7 0007 **********************************************
F7 0007 JOB ARISBDID IS OPTIONAL.
F7 0007 WOULD YOU LIKE TO RUN IT? {Y|N-Default}
F7-0007
7 y
F4 0001 1Q47I   F4 ARISBDID 00300 FROM (HEHA) , TIME=11:36:54, TKN=000001AD
F4 0004 // JOB ARISBDID -- DBNAME DIRECTORY SERVICE GENERATION
F4 0004 * *******************************************************************
F4 0004 *                                                                   *
F4 0004 *  THIS JCL EXECUTES STEPS TO GENERATE THE DBNAME DIRECTORY SERVICE *
F4 0004 *  ROUTINE (ARICDIRD.PHASE).                                        *
F4 0004 *                                                                   *
F4 0004 *  STEP 1 EXECUTES THE PROCEDURE ARICCDID FOR MIGRATION             *
F4 0004 *  OR ARICBDID FOR INSTALLATION TO READ THE DBNAME                  *
F4 0004 *  DIRECTORY SOURCE MEMBER ARISDIRD.A FROM THE PRODUCTION LIBRARY,  *
F4 0004 *  AND GENERATES THE ASSEMBLER VERSION OF ARISDIRD ON SYSPCH.       *
F4 0004 *                                                                   *
F4 0004 * *******************************************************************
F4 0004 * *******************************************************************
F4 0004 * ARISBDID  STEP 1 -- BUILD ASSEMBLER VERSION OF ARISDIRD           *
F4 0004 * *******************************************************************
F4 0004 * SQL/DS DBNAME DIRECTORY BUILT SUCCESSFULLY
F4 0004 EOJ ARISBDID  MAX.RETURN CODE=0000
F7 0007 **********************************************
F7 0007 * Job  ARISBDID  executed successfully
F7 0007 **********************************************
F7 0007
F7 0007 ************ JOB ARIS75BD ********************
F7 0007 * LINK EDIT DB2 SERVER ONLINE SUPPORT COMPONENTS
F7 0007 **********************************************
F7 0007 JOB ARIS75BD IS OPTIONAL.
F7 0007 WOULD YOU LIKE TO RUN IT? {Y|N-Default}
F7-0007
7 y
F4 0001 1Q47I   F4 ARIS75BD 00301 FROM (HEHA) , TIME=11:37:07, TKN=000001AE
F4 0004 // JOB ARIS75BD     LINK EDIT DB2 FOR VSE ONLINE SUPPORT COMPONENTS
F4 0004 * ***********************************************************
F4 0004 * ARIS090D: LINK EDIT SQL/DS ONLINE RESOURCE ADAPTER CONTROL
F4 0004 * ***********************************************************
```

```
F4 0004 * ************************************************************
F4 0004 * ARIS140D: LINK EDIT ISQL
F4 0004 * ************************************************************
F4 0004 * ************************************************************
F4 0004 * ARIS150D: LINK EDIT ISQL ITRM TERMINAL TRANSACTION
F4 0004 * ************************************************************
F4 0004 * ************************************************************
F4 0004 * ARIS160D: LINK EDIT ISQL ITRM TERMINAL EXTENSION PROGRAM
F4 0004 * ************************************************************
F4 0004 EOJ ARIS75BD  MAX.RETURN CODE=0000
F4 0004 EOJ NO NAME
F4 0001 1Q34I   F4 WAITING FOR WORK
F7 0007 **********************************************
F7 0007 * Job  ARIS75BD  executed successfully
F7 0007 **********************************************
F7 0007
F7 0007 ************ JOB ARIS75DD ********************
F7 0007 * DATABASE DBGEN AND SET UP
F7 0007 **********************************************
F4 0001 1Q47I   F4 ARIS75DD 00302 FROM (HEHA) , TIME=11:37:20, TKN=000001AF
F4 0004 // JOB ARIS75DD              DATABASE DBGEN AND SET UP
F4 0004 * ************************************************************
F4 0004 * ARIS75SL: DB2 SERVICE/PRODUCTION LIBRARY DEFINITION
F4 0004 * ************************************************************
F4 0004 * ************************************************************
F4 0004 * ARIS75DB: DB2 STARTER DATABASE IDENTIFICATION
F4 0004 * ************************************************************
F4 0004 * ************************************************************
F4 0004 * ARIS030D: GENERATE THE STARTER DATABASE
F4 0004 * ************************************************************
F4 0004 ARI0025I The program ARISQLDS is loaded at 400078.
F4 0004 ARI0025I The program ARICMOD is loaded at 564D80.
F4 0004 ARI0025I The program ARIXSXR is loaded at 581C00.
F4-0004 ARI0919D Database generation invoked.
                The database will be formatted and the original
                database destroyed.

                Enter either:
                  DBGEN   to continue, or
                  CANCEL  to cancel.
4 dbgen
F4 0004 System identification at DB generation = DB2 VSE & VM 7.5
F4 0004
:
F7 0007 **********************************************
F7 0007 * Job  ARIS75DD  executed successfully
F7 0007 **********************************************
F7 0007
F7 0007 ************ JOB ARIS75ED ********************
F7 0007 * INSTALL DATABASE COMPONENTS (ISQL, FIPS FLAGGER)
F7 0007 **********************************************
F7 0007 JOB ARIS75ED IS OPTIONAL.
F7 0007 WOULD YOU LIKE TO RUN IT? {Y|N-Default}
F7-0007
7 y
F4 0001 1Q47I   F4 ARIS75ED 00303 FROM (HEHA) , TIME=11:45:41, TKN=000001B0
F4 0004 // JOB ARIS75ED     INSTALL DATABASE COMPONENTS
F4 0004 * ************************************************************
F4 0004 * ARIS75SL: DB2 SERVICE/PRODUCTION LIBRARY DEFINITION
F4 0004 * ************************************************************
F4 0004 * ************************************************************
F4 0004 * ARIS75DB: DB2 STARTER DATABASE IDENTIFICATION
F4 0004 * ************************************************************
F4 0004 * ************************************************************
F4 0004 * ARIS080D: GRANT SCHEDULE AUTHORITY TO DBDCCICS
F4 0004 * ************************************************************
:
F4 0001 1Q34I   F4 WAITING FOR WORK
F7 0007 **********************************************
F7 0007 * Job  ARIS75WD  executed successfully
F7 0007 **********************************************
```

```
F7 0007 ************ JOB ARIS75HZ ********************
F7 0007 * ENLARGE HELP TEXT DBSPACE
F7 0007 **********************************************
F7 0007 JOB ARIS75HZ IS OPTIONAL.
F7 0007 WOULD YOU LIKE TO RUN IT? {Y|N-Default}
F7-0007
7 n
F7 0007
F7 0007 ************ JOB ARIS75JZ ********************
F7 0007 * INSTALL LANGUAGE
F7 0007 **********************************************
F7 0007 JOB ARIS75JZ IS OPTIONAL.
F7 0007 WOULD YOU LIKE TO RUN IT? {Y|N-Default}
F7-0007
7 n
F7 0007
F7 0007 ************ JOB ARIS75FD ********************
F7 0007 * GRANT SCHEDULE AUTHORITY
F7 0007 **********************************************
F7 0007 JOB ARIS75FD IS OPTIONAL.
F7 0007 WOULD YOU LIKE TO RUN IT? {Y|N-Default}
F7-0007
7 y
F4 0001 1Q47I   F4 ARIS75FD 00305 FROM (HEHA) , TIME=11:47:09, TKN=000001B1
F4 0004 // JOB ARIS75FD     GRANT SCHEDULE FOR DFHSIT APPLID
F4 0004 * *********************************************************
F4 0004 * ARIS75PL: DB2 PRODUCTION LIBRARY DEFINITION
F4 0004 * *********************************************************
F4 0004 * *********************************************************
F4 0004 * ARIS75DB: DB2 STARTER DATABASE IDENTIFICATION
F4 0004 * *********************************************************
F4 0004 * *************************************************************
F4 0004 * ARISDBSD: EXECUTE THE DBS UTILITY IN SQL/DS SINGLE USER MODE
F4 0004 * *************************************************************
:
F7 0007
F7 0007 ************ JOB ARIS6ASD ********************
F7 0007 * SQL ASSEMBLER SAMPLE PROGRAM
F7 0007 **********************************************
F7 0007 JOB ARIS6ASD IS OPTIONAL.
F7 0007 WOULD YOU LIKE TO RUN IT? {Y|N-Default}
F7-0007
7 y
F7 0001 1Q47I   F7 ARIS6ASD 00318 FROM (HEHA) , TIME=12:33:17, TKN=000001B2
F7 0007 // JOB ARIS6ASD SQL ASSEMBLER SAMPLE PROGRAM
F7 0007 * STEP 1 - PREP
F7 0007 1T20I  SYS079 HAS BEEN ASSIGNED TO X'FED' (PERM)
F7 0007 1T20I  SYSPCH HAS BEEN ASSIGNED TO X'FED' (PERM)
F7 0007 * STEP 2 - ASSEMBLE
F7 0007 1T20I  SYSIPT HAS BEEN ASSIGNED TO X'FEC' (PERM)
F7 0007 * STEP 3 - LINK EDIT
F7 0007 * STEP 4 - EXECUTE THE SAMPLE PROGRAM
F7 0007 EOJ ARIS6ASD  MAX.RETURN CODE=0000
F7 0007 ************ JOB ARIS6CD ********************
F7 0007 * SQL C/370 SAMPLE PROGRAM
F7 0007 **********************************************
F7 0007 JOB ARIS6CD IS OPTIONAL.
F7 0007 WOULD YOU LIKE TO RUN IT? {Y|N-Default}
F7-0007
7 n
F7 0007
F7 0007 ************ JOB ARIS6FTD ********************
F7 0007 * SQL FORTRAN SAMPLE PROGRAM
F7 0007 **********************************************
F7 0007 ************ JOB ARIS6PLD ********************
F7 0007 * SQL PL/I SAMPLE PROGRAM
F7 0007 **********************************************
F7 0007 JOB ARIS6PLD IS OPTIONAL.
F7 0007 WOULD YOU LIKE TO RUN IT? {Y|N-Default}
F7-0007
7 n
```

```
F7 0007 INSTALLATION STEP HAS ENDED SUCCESSFULLY....
F7 0007 PLEASE READ THE PROGRAM DIRECTORY AND PERFORM NECESSARY
F7 0007 STEPS MANUALLY.
F7 0007 EOJ DB2JMGR   MAX.RETURN CODE=0000
```

Step 4: Set Up for DRDA Support

This step, together with Steps 5 and 6, prepares your z/VSE environment for use with the DB2-based connector.

To create the definitions required for the DRDA setup, release job **DB2DRDA** placed by skeleton SKDB2VAR into the VSE/POWER reader queue.

```
* $$ JOB JNM=DB2DRDA,CLASS=R,DISP=L,NTFY=YES
* $$ LST CLASS=Q,DISP=H
// JOB DB2DRDA ACTIVATE DRDA SERVER SUPPORT
* **********************************************************
*  LINK EDIT RDS WITH DRDA SERVER SUPPORT
* **********************************************************
// LIBDEF *,SEARCH=PRD2.DB2750
// LIBDEF PHASE,CATALOG=PRD2.DB2750
// OPTION CATAL
 INCLUDE ARISLKRA
// EXEC  PGM=LNKEDT,PARM='MSHP,AMODE=31,RMODE=ANY'
/*
/&
* $$ EOJ
```

Step 5: Set Up Stored Procedure Server and Define to DB2

Step 5 is divided into:

- "Step 5.1: Set Up the Stored Procedure Server"
- "Step 5.2: Define Stored Procedure Server to DB2" on page 94

Step 5.1: Set Up the Stored Procedure Server

The *Stored Procedure Server* runs per default in a dynamic partition of class R with a partition size of 8MB. You can, however, configure the job to run in another partition that has a larger partition size (8MB is the minimum recommended).

Use skeleton **SKDB2SPS** (located in VSE/ICCF library 59) to catalog the startup job **SPSERV01** for the *Stored Procedure Server* into PRD2.DB2750. This job is shown below.

```
$$ JOB JNM=DB2SPSCA,CLASS=0,DISP=D,NTFY=YES
$$ LST CLASS=Q,DISP=H
// JOB DB2SPSCA CATALOG STORED PROCEDURE SERVER JOB
* ****************************************************************  C
* *                                                              *  C
* *  THIS JOB WILL CATALOG THE JOB THAT WILL RUN IN THE          *  C
* *  STORED PROCEDURE SERVER PARTITION.                          *  C
* *  SERVER PARTITION SHOULD BE A DYNAMIC PARTITION OF AT LEAST  *  C
* *  8MB IN SIZE, DEFAULT IS CLASS R.                            *  C
* *  MODIFY JCL AS FOR YOUR NEEDS. CLASS IN THE POWER PUN CARD   *  C
* *  IS THE PARTITION WHERE THE SERVER RUNS.                     *  C
* *  THE STORED PROCEDURE LIBRARY PRD2.DB2STP SHOULD BE IN THE   *  C
* *  SEARCH CHAIN.                                               *  C
* ****************************************************************  C
                                                                    C
         AFTER YOU HAVE MODIFIED THE SKELETON ENTER '§DTRSEXIT'     C
         FROM THE EDITOR'S COMMAND LINE.                            C
         THIS MACRO WILL DELETE ALL DESCRIPTIVE TEXT FROM THIS FILE,  C
         BY DELETING ALL LINES WHICH ARE MARKED WITH THE CHARACTER C  C
         IN COLUMN 71.                                              C
                                                                    C
```

```
// EXEC  LIBR,PARM='MSHP'
   ACCESS  SUBLIB = PRD2.DB2750
CATALOG SPSERV01.A        EOD=&&          REPLACE=YES
. $$ PUN JNM=SPSERV01,DISP=I,CLASS=R
// JOB SPSERV01    START DB2 STORED PROCEDURE SERVER 01
// OPTION NODUMP,NOSYSDUMP
* // EXEC PROC=ARIS75SL
// ASSGN SYS098,SYSPCH
// LIBDEF *,SEARCH=(PRD2.DB2STP,PRD2.DB2750,PRD2.SCEEBASE,PRD1.BASE)
   ON $RC > 0 GOTO END
// EXEC  PGM=ARISPRC,SIZE=1M
/.END
/*
/&
&&
/*
/&
$$ EOJ
```

A *Stored Procedure Server* is always dedicated to a particular *DB2 Server for VSE* which starts an associated *Stored Procedure Server* during system startup.

Step 5.2: Define Stored Procedure Server to DB2

You can define the Stored Procedure Server SPSERV01 using:

- This DB2 command:

```
CREATE PSERVER SPSERV01 AUTOSTART YES
```

- This batch job, which you can run at this point in the installation:

```
* $$ JOB JNM=PSERVER,CLASS=0,DISP=D
* $$ LST CLASS=W,DISP=H,DEST=(,xxxxx)
// JOB PSERVER
// LIBDEF PROC,SEARCH=(PRD2.DB2750)
// DLBL SQLGLOB,'DB2.SQLGLOB.MASTER',,VSAM,CAT=DB2UCAT,DISP=(OLD,KEEP)
// EXEC PROC=ARIS75PL       *-- DB2 PROD. LIBRARY ID PROC
// EXEC PROC=ARIS75DB       *-- DB2 DATABASE ID PROC
// EXEC  ARIDBS,SIZE=AUTO
CONNECT SQLDBA IDENTIFIED BY SQLDBAPW;
CREATE PSERVER SPSERV01 AUTOSTART YES;
/*
/&
* $$ EOJ
```

Step 6: Set Up for Stored Procedures

Stored procedures need to be catalogued into the system library PRD2.DB2STP.

You must compile your stored procedures with the *reentrant* parameter. For an example of how to do so in a COBOL program, see the RENT option used in skeleton SKDLICMP (which is located in VSE/ICCF library 59).

To create definitions for Stored Procedures, use the skeletons listed under “Step 7: Customize the DB2-Based Connector for VSAM Data Access” on page 95 and “Step 8: Customize the DB2-Based Connector for DL/I Data Access” on page 95.

Stored Procedures are always dedicated to a particular *Stored Procedure Server*.

Step 7: Customize the DB2-Based Connector for VSAM Data Access

If you want to access VSAM data via the DB2-based connector, follow the steps below to create and define the Stored Procedures. The skeletons are provided in VSE/ICCF library 59.

- Step 7.1: Use skeleton SKCRESTP to create Stored Procedures for VSAM data access and define them to DB2.
- Step 7.2: Use skeleton SKCPSTP to compile and link-edit Stored Procedures written in C for VSAM data access.
- Step 7.3: Use skeleton SKVSSAMP to define a VSE/VSAM cluster and load sample data into it.

Step 8: Customize the DB2-Based Connector for DL/I Data Access

If you want to access DL/I data via the DB2-based connector, you must first install the z/VSE optional program DL/I VSE. Then you can follow the steps below. The skeletons are located in VSE/ICCF library 59.

- Step 8.1: Use skeleton SKDLISMP to define and load a DL/I sample database.
- Step 8.2: Use skeleton SKDLISTP to create DB2 Stored Procedures used for accessing the DL/I sample database.
- Step 8.3: Use skeleton SKDLICMP to compile and linkedit COBOL DB2 Stored Procedures used for accessing the DL/I sample database.
- Step 8.4: Customize CICS TS. To access DL/I data via the DB2-based connector, you must customize a CICS TS - DL/I online system:
  1. Configure your CICS/DLI online system, as described in:
     - Part 6 of the *DL/I Resource Definition and Utilities*.
     - The topic “CICS – DL/I Tables – Requirements” of the *DL/I Resource Definition and Utilities*.
     - Topic “Migrating to DL/I VSE 1.11 and the CICS Transaction Server for VSE/ESA 1.1” of the *DL/I Release Guide*.
  2. Define the sample database STDIDBP and other databases you wish to access, to CICS (using either the CICS FCT or transaction CEDA).
  3. Provide labels for the sample database STDIDBP (// DLBL STDIDBC ...) and other databases you wish to access.
  4. Create a new DL/I online nucleus (DLZACT generation), by including all DL/I online programs and PSBs that you wish to use. The CICS/DLI mirror program DLZBPC00 must be authorized for PSB STBICLG, used for accessing the DL/I sample database. The CICS/DLI mirror program DLZBPC00 must also be authorized for any other PSBs you wish to use for accessing other DL/I databases.
  5. Account for an increased number of concurrent DLZBPC00 mirror tasks in the CICS/DLI online system: you must accordingly adjust the MAXTASK and CMAXTSK parameters in the DLZACT generation.
  6. Load the DL/I exit routine DLZBSEOT into the SVA.
  7. Start an MPS system.

  DLZMPX00 is SVA-eligible, and is used for accessing DL/I data via the *AIBTDLI interface* (see “Overview of the AIBTDLI Interface” on page 403 for an explanation of DLZMPX00 and the AIBTDLI interface). The AIBTDLI interface uses DLZMPX00 from the SVA (if it resides there), or loads DLZMPX00 into partition space and uses it from there.

Step 9: Start DB2, and Start Stored Procedure Server

Run skeleton **SKDB2STR** to place the startup job **DB2START** into the VSE/POWER reader queue. This skeleton is shown below.

```
* $$ JOB JNM=DB2START,CLASS=S,DISP=L,NTFY=YES
* $$ LST CLASS=Q,DISP=H
// JOB DB2START DB2 SERVER STARTUP JOB
// LIBDEF PROC,SEARCH=(PRD2.DB2750,PRD2.DB2STP)
// LIBDEF PHASE,SEARCH=(PRD1.BASE,PRD2.SCEEBASE,PRD2.DB2750,           X
               PRD2.DB2STP)
// EXEC  PROC=ARIS75SL    *-- DB2 PRODUCTION LIBRARY ID PROC
// EXEC  PROC=ARIS75DB    *-- DB2 DATABASE ID PROC
// ASSGN SYS098,SYSPCH    *-- DB2 ENABLE POWER FOR STORED PROC HANDLER
// EXEC  ARISQLDS,SIZE=AUTO,PARM='TCPPORT=446,NCUSERS=05,DBNAME=SQLDS, X
               DSPLYDEV=B,RMTUSERS=10'
/*
/&
* $$ EOJ
```

You start the *DB2 Server for VSE* by releasing DB2START. The *DB2 Server for VSE* subsequently starts the *Stored Procedure Server* via startup job SPSERV01 retrieved from PRD2.DB2750. You may include DB2START in SKJCL1 for automatic startup processing (also refer to "Step 1: Customize CICS TS" on page 84).

DB2START requires that TCP/IP is running. To ensure that TCP/IP is running, you can insert the following job step in DB2START, before the EXEC ARISQLDS statement:

```
 // EXEC REXX=IESWAITR,PARM='TCPIP00'
 /*
```

where *TCPIP00* is the name of your TCP/IP startup job.

For further startups, you might consider opening the Sample database by inserting the CIRB transaction into your CICS TS startup job, as shown below:

```
// EXEC DFHSIP,SIZE=DFHSIP,PARM='SIT=C3,START=COLD,SEC=NO,STATRCD=OFF,S*
               VA=NO,NEWSIT=YES,DSALIM=8M,EDSALIM=30M,SI',DSPACE=2M,OS3*
               90
/*
CIRB PASSWORD,8,XXX03,1,GER,PRODDBI
/*
```

Step 10: Install DB2 Connect and Establish Client-Host Connection

To finally establish a connection from the client to the z/VSE host, you must install DB2 Connect Version 6 Release 1 or later, on your physical/logical *middle-tier* (if not already installed) of a 3-tier environment as described in "Overview of 3-Tier Environments" on page 8. You must then *configure* DB2 Connect to enable it to access DB2 data stored on the z/VSE host. To do so, you can use either use the *Client Configuration Assistant* (CCA), or the DB2 command-line interface, *as described below*.

The sample database **sqlds** is stored on the z/VSE host, and is used together with the DB2-Based Connector. To define the **sqlds** database to DB2 Connect on the physical/logical middle-tier so that it has the alias **db2vsewm**, you must:

1. Define the communication protocol between DB2 Connect on the physical/logical middle-tier, and the database (**sqlds**) residing on DB2 Server for VSE. To do so, you define a *node*, as follows:

   ```
   db2 catalog protocol node nodename remote ip-addr server port-nr
   ```

   For the DB2-Based Connector samples (the DL/I applet and VSAM applet), you would enter a command such as:

   ```
   db2 catalog tcpip node tcpvse remote 9.111.122.33 server 446
   ```
2. Define the entry for the Database Connection Services (DCS). To do so, use this command:

   ```
   db2 catalog dcs database dcs-name as vse-dbname
   ```

   For the DB2-Based Connector samples (the DL/I applet and VSAM applet) you would enter a command such as:

   ```
   db2 catalog dcs database dcsdb as sqlds
   ```
3. Define the *alias* used by the DB2-Based Connector samples. To do so, use this command:

   ```
   db2 catalog dcs-name  as vse-alias-dbname at node nodename authentication dcs
   ```

   For the DB2-Based Connector samples (the DL/I applet and VSAM applet) you would enter a command such as:

   ```
   db2 catalog dcsdb as db2vsewm at node tcpvse authentication dcs
   ```

After you have completed the definition of the database **sqlds** to DB2 Connect, you must now execute the *bind* step:

```
db2 bind path@ddcsvse.lst blocking all sqlerror continue messages msg-file
grant public
```

For example, for Windows XP/Vista/7 you would enter:

```
db2 bind db2\bnd\@ddcsvse.lst blocking all sqlerror continue messages log.msg
grant public
```

For detailed information on how to install and customize DB2 Connect, refer to the:

- *DB2 Connect User's Guide*, SC09-2838.
- DB2 home page at: `http://www.ibm.com/db2` which has links to most publications covering DB2. By selecting **Library** and then **DB2 Publications** you can browse most publications (in PDF, Postscript, and/or HTML format) concerning both DB2 Server for VSE and DB2 Connect.

Chapter 11. Configuring the VSAM-Via-CICS Service

Until VSE/ESA 2.6, if a VSAM cluster was opened for update by CICS, the VSE Connector Server and DB2 Stored Procedures could only access the same VSAM cluster in read-only mode, *unless* the VSAM shareoption 4 was used. Using VSAM shareoption 4 resulted in performance degradation.

From VSE/ESA 2.6 onwards and using the *VSAM-via-CICS service*, the VSE Connector Server and DB2 Stored Procedures can access VSAM data *without* the restriction that you must use VSAM shareoption 4.

VSAM clusters must be opened *only once* by CICS. The sharing problem no longer exists, and use of VSAM shareoption 4 is no longer required.

This chapter describes how you can take advantage of this performance improvement.

It contains these main topics:

- “Configuring the IBM-Supplied CICS System”
- “Configuring a Further CICS System for VSAM-Via-CICS” on page 100
- “How the VSAM-Via-CICS Service Works” on page 101
- “CICS Transactions for Use with VSAM-Via-CICS” on page 101

Configuring the IBM-Supplied CICS System

These CICS programs are included for accessing VSAM via CICS:
- IESCVSRV (server task)
- IESCVMIR (mirror task)
- IESCVSTA (start transaction)
- IESCVSTI (internal start transaction)
- IESCVSTP (stop transaction)

These file types are supported by the VSAM-via-CICS service:
- ESDS
- KSDS
- RRDS
- KSDS-PATH
- ESDS-PATH

To use the VSAM-via-CICS service, you must therefore:

1. Define the VSAM clusters that you wish to access to CICS. To do so, use the `CEDA DEFINE` transaction. Ensure that the VSAM clusters are defined (using `CEDA DEFINE`) so that:
   - All VSAM clusters are enabled.
   - For read-only access, the VSAM clusters are `readable` and `browsable`.
   - For write access, the VSAM clusters are `addable`, `updateable`, and `deleteable`.
2. Ensure that the module IESCVSVA.PHASE is loaded in the SVA, before you start:
   - The VSAM-via-CICS service
   - CICS

© Copyright IBM Corp. 2000, 2013

3. Ensure the VSAM-via-CICS service is active. The IBM-supplied CICS system is configured so that this service will be started *automatically*. If you wish to start this service *yourself*, use transaction ICVA to do so. To stop the VSAM-via-CICS service, use transaction ICVP.
4. Ensure that the applications that are to access VSAM clusters via CICS (either VSE Connector Client applications, or applications that call DB2 Stored Procedures) use these naming conventions:
   - Catalog File ID must be named as:

     ```
     #VSAM.#CICS.CICS applid
     ```

     For example, #VSAM.#CICS.DBDCCICS
   - Cluster File ID must be the same as the one you used for CICS (consisting of 7-characters).

   Here is an example of how to use these naming conventions. Assume a VSAM cluster exists with the name `MY.TEST.CLUSTER`, and that this cluster resides in the VSAM catalog `MY.USER.CATALOG`. The file is defined as `MYTEST` in the CICS system that has applid DBDCCICS.

   To access file `MYTEST` from either a VSE Connector Client application or an application calling DB2 Stored Procedures, you would use these mapping names:

   ```
   Catalog File ID: #VSAM.#CICS.DBDCCICS
   Cluster File ID: MYTEST
   ```

The only change that you therefore must make to your existing programs (VSE Connector Client applications, or applications calling DB2 Stored Procedures) is to ensure that these programs use the naming conventions for VSAM mapping, as described above.

To optimize performance, you can:
- Access VSAM clusters in batch for *read* commands, using the original VSAM name.
- Access VSAM clusters using the VSAM-via-CICS service for *write* commands, using the VSAM-via-CICS service name.

Configuring a Further CICS System for VSAM-Via-CICS

From VSE/ESA 2.6 onwards, each shipped CICS system is configured *by default* so that the VSAM-via-CICS service is active. Therefore, you are not required to perform any customization activities to other CICS systems shipped from VSE/ESA 2.6 onwards.

However, if you wish to configure a CICS system *that was shipped before VSE/ESA 2.6* to make use of the VSAM-via-CICS service, you must:
1. Define the following programs (providing they are not already defined) in your CICS in the same way as they are defined in the IBM-supplied CICS:
   - IESCVSRV (server task)
   - IESCVMIR (mirror task)
   - IESCVSTA (start transaction)
   - IESCVSTI (internal start transaction)
   - IESCVSTP (stop transaction)
2. Define the following transactions (providing they are not already defined) in your CICS in the same way as they are defined in the IBM-supplied CICS:
   - ICVS
   - ICVM

- ICVA
- ICVP

(For details of these transactions, see "CICS Transactions for Use with VSAM-Via-CICS").

3. If you wish the VSAM-via-CICS service to be *automatically* started, you must:
   a. Add this statement to the CICS PLT table DFHPLTPI (as the *last* statement in the table):
      ```
      DFHPLT TYPE=ENTRY, PROGRAM=IESCVSTI
      ```
   b. Add this statement to the CICS PLT table DFHPLTSD (as the *first* statement in the table):
      ```
      DFHPLT TYPE=ENTRY, PROGRAM=IESCVSTP
      ```
4. Ensure that the IESCVSVA.PHASE is loaded into the SVA (the batch-side support for VSAM access via CICS resides in $IESCVBA.PHASE), using load list $SVACONN.

How the VSAM-Via-CICS Service Works

The VSAM-via-CICS service works in this way:

1. A VSE Connector Client application or an application calling a DB2 Stored Procedure issues a VSAM-access request.
2. The request is sent to the batch partition where the VSE Connector Server or DB2 Stored Procedure is running.
3. The request is forwarded to CICS via XPCC (cross-partition communication).
4. The VSAM-via-CICS service running within the CICS Transaction Server executes the request (for example, reads a record) and passes the data back to the batch partition where the VSE Connector Server or DB2 Stored Procedure is running.
5. The VSE Connector Server or DB2 Stored Procedure returns the data to the application.

CICS Transactions for Use with VSAM-Via-CICS

The following CICS transactions have been provided for use with VSAM-via-CICS service (and are pre-defined in the VSE/ESA 2.6 system):

ICVS The server transaction. Uses program IESCVSRV.

ICVM The mirror transaction. Uses program IESCVMIR.

ICVA You can use this transaction to start the VSAM-via-CICS service. Uses program IESCVSTA.

ICVP You can use this transaction to stop the VSAM-via-CICS service. Uses program IESCVSTP.

Chapter 12. Mapping VSE/VSAM Data to a Relational Structure

This chapter describes how VSE/VSAM data is mapped to a relational structure, and then the four methods that you can use to do so.

It contains these main topics:

- "Introduction to Mapping VSE/VSAM Data"
- "How VSAM Maps Are Structured" on page 104
- "How Maps Are Stored on the z/VSE host" on page 104
- "Defining a Map Using RECMAP" on page 105
- "Defining a Map Using the Sample Applet" on page 105
- "Defining a Map Using a Java Application" on page 106
- "Defining a Map Using the VSAM MapTool" on page 112

Introduction to Mapping VSE/VSAM Data

You must map VSAM records to a relational structure, if you wish to access VSAM data using:

- A DB2 Stored Procedure via the VSAM CLI (Call Level Interface), as described in "Using DB2 Stored Procedures to Access VSAM Data" on page 395.
- VSE Java Beans, as described in "Contents of the VSE Java Beans Class Library" on page 143.

In the past, VSAM data was mainly accessed using application programs that understood the internal structure of the VSAM data: the record layout was represented by data structures within the application programs. The disadvantages of using this method are:

- There is no way to share a given record layout with other applications.
- If the record layout changes, the application program must also be changed.
- The data structure is dependent on the programming language (for example COBOL, Assembler, or PL/I).
- Formatted data reports have to be created on the operating-system platform on which the application programs run.

However, the development of *e-business applications* as described in this publication, requires:

- A sharing of data representation across operating-system platforms.
- Easy access to data representations from different applications.
- That data representation and data display are independent of operating-system platform and programming language.

z/VSE therefore provides you with two methods of mapping your VSAM data so it can be used within e-business applications:

- Creating a *data map* (referred to simply as a *map)* for a given VSAM record or record type. A map splits the VSAM record into columns and their data fields, that have a name, a length, a datatype, and an offset within the record.
- Creating a *data view* (referred to simply as a *view)* that contains a subset of the fields contained within a map. A view always points to a subset of the data fields of a given VSAM map. Therefore, if you change a VSAM map, the views

© Copyright IBM Corp. 2000, 2013

that use this map will also be affected. For example, deleting a map will also delete all views of this map. You can use views to give different user groups different views of the same data (for example, to hide some information from specific users).

How VSAM Maps Are Structured

The mapping definitions have the following *hierarchical* structure:

```
CATALOG       (MY.USER.CATALOG)
 CLUSTER      (MY.DATA.CLUSTER)
  MAP         (Map One)
   COLUMN     (Name,      Ofs=0,Len=10,Type=String,Description=xxxxx)
   COLUMN     (Street,    Ofs=10,Len=20,Type=String,Description=xxxxx)
   ...
   VIEW       (View One)
     COLUMN   (PersonInfo, Ref=Name,Description=xxxxx)
     COLUMN   (Address,    Ref=Street,Description=xxxxx)
     ...
   VIEW       (View Two)
   ...
  MAP         (Map Two)
  ...
```

Figure 20. Hierarchical Structure of VSAM Maps

A data field represents one specific column of a VSAM record. It is always part of a given map or view. It consists of:

- a name (the column name)
- a length
- a datatype
- an offset within the record
- a description

When an application accesses VSAM data using a map, only the *map's* fields are used.

You can also specify filters when accessing VSAM data using a map. For example, you might display only those records where a given column (data field) matches a given filter string. You can use filters in any Java program (Java applications, Java servlets, Java applets, or Enterprise Java Beans).

Note: When you define a map for a cluster, no check is made to see if a cluster exists. Therefore you must take steps to ensure that your mapping definitions are always in synchronization with your VSAM clusters.

How Maps Are Stored on the z/VSE host

On the host, the maps of all VSAM clusters are stored in one single VSAM file, `VSE.VSAM.RECORD.MAPPING.DEFS`. *This file is created automatically during the initial installation of z/VSE.*

The short name for this VSAM file is IESMAPD, and the cluster for this file is stored in `VSESP.USER.CATALOG`.

Figure 21 is an example of a job used to define the cluster for file `VSE.VSAM.RECORD.MAPPING.DEFS`. You can find this example job in the member VSAMDEFS.Z in Library IJSYSRS.SYSLIB.

Notes:

1. You should not normally need to run this job, since it runs automatically during z/VSE installation.
2. You should not delete or move this file.

```
* $$ JOB JNM=DEFINE,CLASS=0,DISP=D
// JOB  DEFINE FILE
// EXEC IDCAMS,SIZE=AUTO
         DEFINE CLUSTER (NAME (VSE.VSAM.RECORD.MAPPING.DEFS) -
         RECORDS(2000,1000) -
         SHAREOPTIONS (2) -
         RECORDSIZE (284    284   ) -
         VOLUMES (DOSRES SYSWK1 ) -
         NOREUSE -
         INDEXED -
         FREESPACE (15 7) -
         KEYS (222 0     ) -
         NOCOMPRESSED -
         TO (99366  )) -
         DATA (NAME (VSE.VSAM.RECORD.MAPPING.DEFS.@D@) -
         CONTROLINTERVALSIZE (4096  )) -
         INDEX (NAME (VSE.VSAM.RECORD.MAPPING.DEFS.@I@)) -
         CATALOG (VSESP.USER.CATALOG)
IF LASTCC NE 0 THEN CANCEL JOB
/*
/&
* $$ EOJ
```

Figure 21. Job To Define the Cluster for VSE.VSAM.RECORD.MAPPING.DEFS

Defining a Map Using RECMAP

This method of defining a map for a VSAM cluster uses the IDCAMS **RECMAP** command. Using RECMAP, you can also delete, change (alter), or list the contents of a map or view.

Related Topics:

- The syntax of the RECMAP command is described in the *VSE/VSAM Commands*, SC33-8315.
- A practical example of how RECMAP can be used is provided in “4. Define the VSAM Data Cluster” on page 204.

Defining a Map Using the Sample Applet

This method of defining a map for a VSAM cluster uses *an applet* that runs on a *Web browser.*

A sample applet for used for defining maps is contained in **VsamMappingApplet.html** of the z/VSE e-business Connectors online documentation, and is described in “Running the Sample Data-Mapping Applet” on page 192.

Defining a Map Using a Java Application

This method of defining a map for a VSAM cluster uses a *Java application* that runs on *a workstation.*

The example described here is for a simple Java application that creates a map for a given VSAM cluster, and also defines two different views. The example is taken from the *z/VSE e-business Connectors* online documentation.

If you require the online version of this example, refer to *VsamDataMapping.html*. The source code is contained in Java source file *VsamDataMapping.java*, which is located in the **com.ibm.vse.samples** directory, that is positioned below the samples directory of your connector client installation (described in "Installing the VSE Connector Client" on page 24).

The following steps describe how you code a Java application that defines a map for a VSAM cluster.

1. *Create a Map for the VSAM Cluster:*

   To create a map, you must first instantiate a local object of the *VSEVsamMap* class. The creation of the map definition on the host also requires a call to the *create()* method.

   To use an existing map, you must first instantiate a local object of class *VSEVsamMap* and then use the *isExistent()* method to check if this map exists on the host.

   Note: Be aware that distributed objects are being used. Therefore, when local objects of class *VSEVsamMap* and *VSEVsamView* are created, this does not automatically imply that the related definitions are created on the host.

   Create a local map object and check whether it exists on the host. If it does not exist, create it on the host using the *create()* method. Put the code into try/catch blocks, since the *create()* and *isExistent()* methods can produce exception errors. Here is an example of how to create a local VSAM map object:

```
...
public static void main(String argv[ ]) throws IOException
{
   VSESystem system;
   VSEVsamMap map;
   VSEVsamView employeeView=null, managerView=null;
   VSEVsamField dataField;
   ...
   // Create VSESystem and connect
   ...

   // Create map ...
   String catalogID = "VSESP.USER.CATALOG";
   String clusterID = "VSAM.DISPLAY.DEMO.CLUSTER";
   try
   {
     System.out.println("Creating map MYMAP ...");
     map = new VSEVsamMap(system, catalogID, clusterID, "MYMAP");
     if (!(map.isExistent()))
     {
        System.out.println("  Map does not exist on host. Create it ...");
        map.create();
     }
     else
        System.out.println("  Map already exists on host. Continue ...");
   }
   catch (Exception e)
   {
      System.out.println("  Exception when creating map: ");
      System.out.println(e);
      return;
   }
   ... (Continued)
```

Figure 22. Example of Creating a Local VSAM Map Object

2. *Create Data Fields for the Map*

   Data fields of a given view of a map are simply references to the fields of the map. But the same field can have different names in different views. Deleting a field of a map, for example, will also make this field unavailable for all views. Deleting the whole map will also delete all views of this map. However, your local view objects *are not notified* that their map has been deleted. When the next access is made to a field that belongs to a view, an exception will be generated.

   Now you must add fields to the map, which describe the record's internal structure. Note that these field names must be in *uppercase* (otherwise they will be translated to uppercase when sending the request to the host). Whereas creating a *VSEVsamField* object is a local operation only, the *addField()* method sends this request to the host. Here is an example of how to create data fields for a map:

```
System.out.println("   ");
System.out.println("Adding fields to the map...");
String[ ] fields = {"LAST NAME", "FIRST NAME", "DEPARTMENT", "AGE"};
int[ ] types = {VSEVsamMap.TYPE_STRING, VSEVsamMap.TYPE_STRING,
               VSEVsamMap.TYPE_UNSIGNED, VSEVsamMap.TYPE_UNSIGNED};
int[ ] lengths = {20, 10, 4, 2};
int[ ] offsets = {0, 20, 30, 34};

for (int i=0;i<fields.length;i++)
{
   try
   {
      dataField = new VSEVsamField(system, fields[i], types[i],
                                   lengths[i], offsets[i]);
      map.addField(dataField);
   }
   catch (Exception e)
   {
      System.out.println("Exception when adding " + fields[i] + "
                                       to the map: ");
   }
}
... (Continued)
```

Figure 23. Example of Creating Data Fields for a Map

3. *Display the Properties of the Map*

   Here is an example of how to display some of the properties of the map you have created:

```
...
/* Now display some properties of the map */
System.out.println("Map properties");
System.out.println("  Map name        : " + map.getName());
System.out.println("  Catalog name    : " + map.getCatalog());
System.out.println("  Cluster name    : " + map.getCluster());
System.out.println("  System name     : " + map.getVSESystem());
System.out.println("  Number of fields : " + new Integer(
                                   (map.getNoOfFields())).toString());
for (int i=0;i<map.getNoOfFields();i++)
{
   System.out.println("  Field " + new Integer(i).toString() + " : " +
                                                map.getFieldName(i));
   System.out.println("   type   = " + new Integer(
                                   (map.getFieldType(i))).toString());
   System.out.println("   length = " + new Integer(
                                  (map.getFieldLength(i))).toString());
   System.out.println("   offset = " + new Integer(
                                  (map.getFieldOffset(i))).toString());
}
... (Continued)
```

Figure 24. Example of Displaying the Properties of a Map

4. *Create the Employee View for the Map*

   The example provided in the z/VSE e-business Connectors online documentation creates two different views within this map: a manager's view (not shown here), and an employee's view. Again, creating the view object is only local, but the *create()* method creates the view on the host.

```
...
System.out.println("   ");
System.out.println("Creating employee's view on the map ...");
try
{
   employeeView = new VSEVsamView(system, catalogID, clusterID,
                               "MYMAP", "EMPLOYEEVIEW");
   if (!(employeeView.isExistent()))
   {
      System.out.println("Employee view does not exist. Create it ...");
      employeeView.create();
   }
   else
      System.out.println("Employee already exists. Continue ...");
}
catch (Exception e)
{
   System.out.println("Exception when creating employeeView: ");
   return;
}
...
...
```

Figure 25. Example of Creating a View for a Map

5. *Add Data Fields to the Employee View*

   In the example we now add fields to the employee view. This uses the same method as adding fields to a map. We also decide that employees are allowed to see only three fields of a VSAM record: the last name, the first name, and the department number. Here is the code for such a scenario:

```
String[ ] empFields = {"LAST NAME", "FIRST NAME", "DEPARTMENT"};
System.out.println("Adding fields to employeeView ...");
for (int i=0;i<emp.Fields.length;i++)
{
     try
     {
        employeeView.addField(empFields[i]);
     }
     catch (Exception e)
     {
        System.out.println("Exception when adding " + empFields[i] + "
                                    to employeeView: ");
     }
}
  ... (Continued)
```

Figure 26. Example of Adding Data Fields to a View

6. *Display Properties of the Employee View*

   Here is an example of how to display the properties of the employee view:

```
/* Now display some properties of the employeeView */
   System.out.println("employeeView properties");
   System.out.println("  View name        : " + employeeView.getName());
   System.out.println("  Catalog name     : " + employeeView.getCatalog());
   System.out.println("  Cluster name     : " + employeeView.getCluster());
   System.out.println("  System name      : " + employeeView.getVSESystem());
   System.out.println("  Number of fields : " + new Integer(
                                       (employeeView.getNoOfFields())).toString());
   for (int i=0;i<employeeView.getNoOfFields();i++)
   {
   System.out.println("  Field " + new Integer(i).toString() + " : " +
                                                    employeeView.getFieldName(i));
      System.out.println("   type   = " + new
                                   Integer((employeeView.getFieldType(i))).toString());
      System.out.println("   length = " + new
                                 Integer((employeeView.getFieldLength(i))).toString());
      System.out.println("   offset = " + new
                                 Integer((employeeView.getFieldOffset(i))).toString());
   }
   ... (Continued)
```

Figure 27. Example of Displaying the Properties of a View

7. *Delete the Map*

   Here the map is deleted, together with its views and fields:

```
/* Now delete the map including all views and fields */
   try
   {
      System.out.println("Deleting map MYMAP ...");
      map.delete();
   }
   catch (Exception e)
   {
      System.out.println("Exception when deleting map:");
      System.out.println(e);
   }
   ...
}
```

Figure 28. Example of How to Delete a Map

Java Console Output Produced By the Sample

The following output was produced on the Java console when running this sample from the Web browser (using file *RunExamples.html*) on the IBM test system.

```
C:\vsecon\samples>java  com.ibm.vse.samples.VsamDataMapping
Please enter your VSE IP address:
9.164.155.95
Please enter your VSE user ID:
sysa
Please enter password:
*****  (password will display in clear)
Creating map MYMAP ...
  Map does not exist on host. Create it ...

Adding fields to the map...
Map properties
  Map name         : MYMAP
  Catalog name     : VSESP.USER.CATALOG
  Cluster name     : VSAM.DISPLAY.DEMO.CLUSTER
  System name      : 9.164.155.95
  Number of fields : 4
  Field 0 : LAST NAME
   type   = 2
   length = 20
   offset = 0
  Field 1 : FIRST NAME
```

```
   type   = 2
   length = 10
   offset = 20
  Field 2 : DEPARTMENT
   type   = 3
   length = 4
   offset = 30
  Field 3 : AGE
   type   = 3
   length = 2
   offset = 34

Creating employee's view on the map ...
Employee view does not exist. Create it ...
Adding fields to employeeView ...
employeeView properties
  View name        : EMPLOYEEVIEW
  Catalog name     : VSESP.USER.CATALOG
  Cluster name     : VSAM.DISPLAY.DEMO.CLUSTER
  System name      : 9.164.155.95
  Number of fields : 3
  Field 0 : LAST NAME
   type   = 2
   length = 20
   offset = 0
  Field 1 : FIRST NAME
   type   = 2
   length = 10
   offset = 20
  Field 2 : DEPARTMENT
   type   = 3
   length = 4
   offset = 30

Creating manager's view on the map ...
Manager view does not exist. Create it ...
Adding fields to managerView ...
Exception when adding MONTHLY SALARY to managerView:
managerView properties
  View name        : MANAGERVIEW
  Catalog name     : VSESP.USER.CATALOG
  Cluster name     : VSAM.DISPLAY.DEMO.CLUSTER
  System name      : 9.164.155.95
  Number of fields : 4
  Field 0 : LAST NAME
   type   = 2
   length = 20
   offset = 0
  Field 1 : FIRST NAME
   type   = 2
   length = 10
   offset = 20
  Field 2 : DEPARTMENT
   type   = 3
   length = 4
   offset = 30
  Field 3 : AGE
   type   = 3
   length = 2
   offset = 34
Deleting map MYMAP ...
Finished.

C:\vsecon\samples>pause
Press any key to continue . . .
```

After you have defined one or more maps for a VSAM cluster (which describe the internal structure of VSAM records), you can display the mapped VSAM data using:

- A DB2 Stored Procedure via the VSAM CLI Interface (described in "Using DB2 Stored Procedures to Access VSAM Data" on page 395)
- The VSE Java Beans (described in "Example of Using VSE Java Beans to Access VSAM Data" on page 156).

Defining a Map Using the VSAM MapTool

This method of defining a map for a VSAM cluster uses the *VSAM MapTool*, which creates a map by parsing a COBOL copybook.

In addition, you can use the VSAM MapTool to:

- Import (receive) a specified map from a specified z/VSE system.
- Export a map to a z/VSE system (that is, send it to VSE).
- Import a map from an XML file.
- Export a map to an XML file.
- Create a Java source file from a specified map. The Java program can obtain all records from the related VSAM file, by using this map.

Figure 29 shows window that the VSAM MapTool displays:

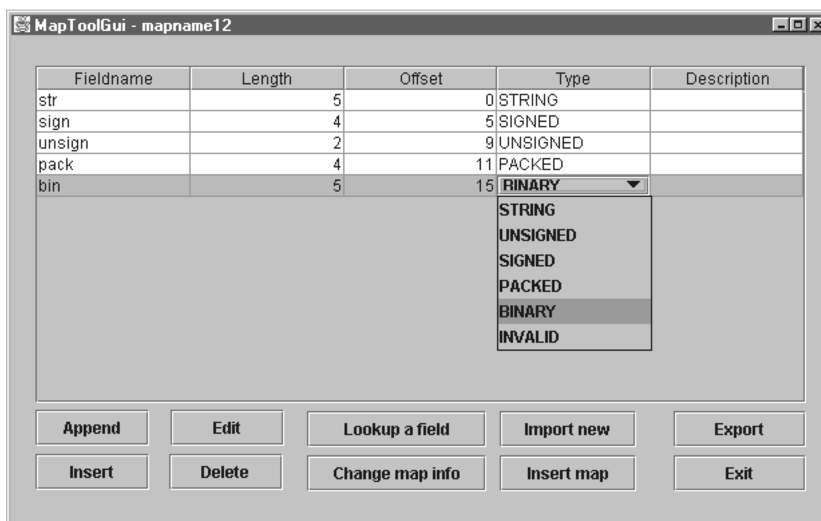


Figure 29. Example of a VSAM MapTool Window

To install the VSAM MapTool you must:

1. Download the file **maptool.zip** from this internet address:

   `http://www.ibm.com/systems/z/os/zvse/downloads/`
2. Run the installation procedure described on the Web site whose address is given above.

The installation package includes many examples of how to use the VSAM MapTool.

Part 3. SYSTEM MONITORING

Chapter 13. Collecting Data via the VSE Monitoring Agent 115
Overview of How VSE Data is Collected 115
Configuring the VSE Monitoring Agent. 117
Maximizing the Security of SNMP Communication 118
Configuring the IBM-Supplied Monitoring Plug-ins 119
Commands Available With the VSE Monitoring Agent 120
Examples of How to Collect Data 121
Using the SNMP Trap Client in Batch Jobs . . . 121
Using the Trap Client API in LE/C Programs. . . 123
Functions Provided by the Trap LE/C interface . . 124
Using the Trap Client API in COBOL and PL/I Programs. 124
Using the Trap Client API in CICS Programs . . . 125
Copybook Used for the Trap COBOL and PL/I Batch interface 126
Return Codes Generated by the Trap Client API 127
Creating Your Own Plug-Ins for the VSE Monitoring Agent 127

Chapter 14. Using GDPS Support for High Availability 129
Overview of How the GDPS Client Is Used . . . 129
Configuring the GDPS Client 130
Starting the GDPS Client 131
Communication Between GDPS Client and GDPS K-System 131
Commands Available With the GDPS Client . . . 132

© Copyright IBM Corp. 2000, 2013

Chapter 13. Collecting Data via the VSE Monitoring Agent

This chapter describes how you can use the *VSE Monitoring Agent* to collect data from your z/VSE systems.

It contains these main topics:

- "Overview of How VSE Data is Collected"
- "Configuring the VSE Monitoring Agent" on page 117
- "Maximizing the Security of SNMP Communication" on page 118
- "Configuring the IBM-Supplied Monitoring Plug-ins" on page 119
- "Commands Available With the VSE Monitoring Agent" on page 120
- "Examples of How to Collect Data" on page 121
- "Using the SNMP Trap Client in Batch Jobs" on page 121
- "Using the Trap Client API in LE/C Programs" on page 123
- "Functions Provided by the Trap LE/C interface" on page 124
- "Using the Trap Client API in COBOL and PL/I Programs" on page 124
- "Using the Trap Client API in CICS Programs" on page 125
- "Copybook Used for the Trap COBOL and PL/I Batch interface" on page 126
- "Return Codes Generated by the Trap Client API" on page 127
- "Creating Your Own Plug-Ins for the VSE Monitoring Agent" on page 127

Note: Throughout this chapter, a short-form of each SNMP member is used that does *not* include the term "SNMP". For example, the SNMP Trap Client API is referred to simply as the *Trap Client API*. Or the SNMP Trap CICS interface is referred to as the *Trap CICS interface*. However, an exception is the *SNMP Trap Client* which has *no short-form*.

Overview of How VSE Data is Collected

IBM has developed an *application framework* to enable you to monitor your z/VSE systems using *standard monitoring interfaces*. This application framework is based upon the common standards TCP/IP and SNMP. It includes an *open interface* which enables you to use your own programs to collect other data.

It is important to monitor your z/VSE systems since core applications (CICS, COBOL, VSAM, and so on) running on the z/VSE host are usually critical to the company's operations. In addition, these core applications usually represent an enormous investment of past resources.

The application framework consists of *three* logical layers:

- A *client*, which can be a workstation, server, wireless telephone, or Personal Digital Assistant (PDA) where monitoring software installed.
- The *VSE Monitoring Agent*, which lies at the center of the application framework and which:
  - processes requests from clients
  - controls access to business logic and data.
- *Plug-ins*, which provide access to:
  - z/VSE data
  - other data via your own programs.

© Copyright IBM Corp. 2000, 2013

As shown in Figure 30, z/VSE provides the:

- VSE Monitoring Agent, which responds to all SNMP Version 1 requests.
- SNMP Trap Client, which can be used for monitoring purposes by sending SNMP Version 1 “traps” to SNMP monitors from batch jobs.
- Trap Client API (IESMTRPA.PHASE), which can be used for monitoring purposes by sending SNMP Version 1 “traps” to SNMP monitors from COBOL, C, and CICS programs.

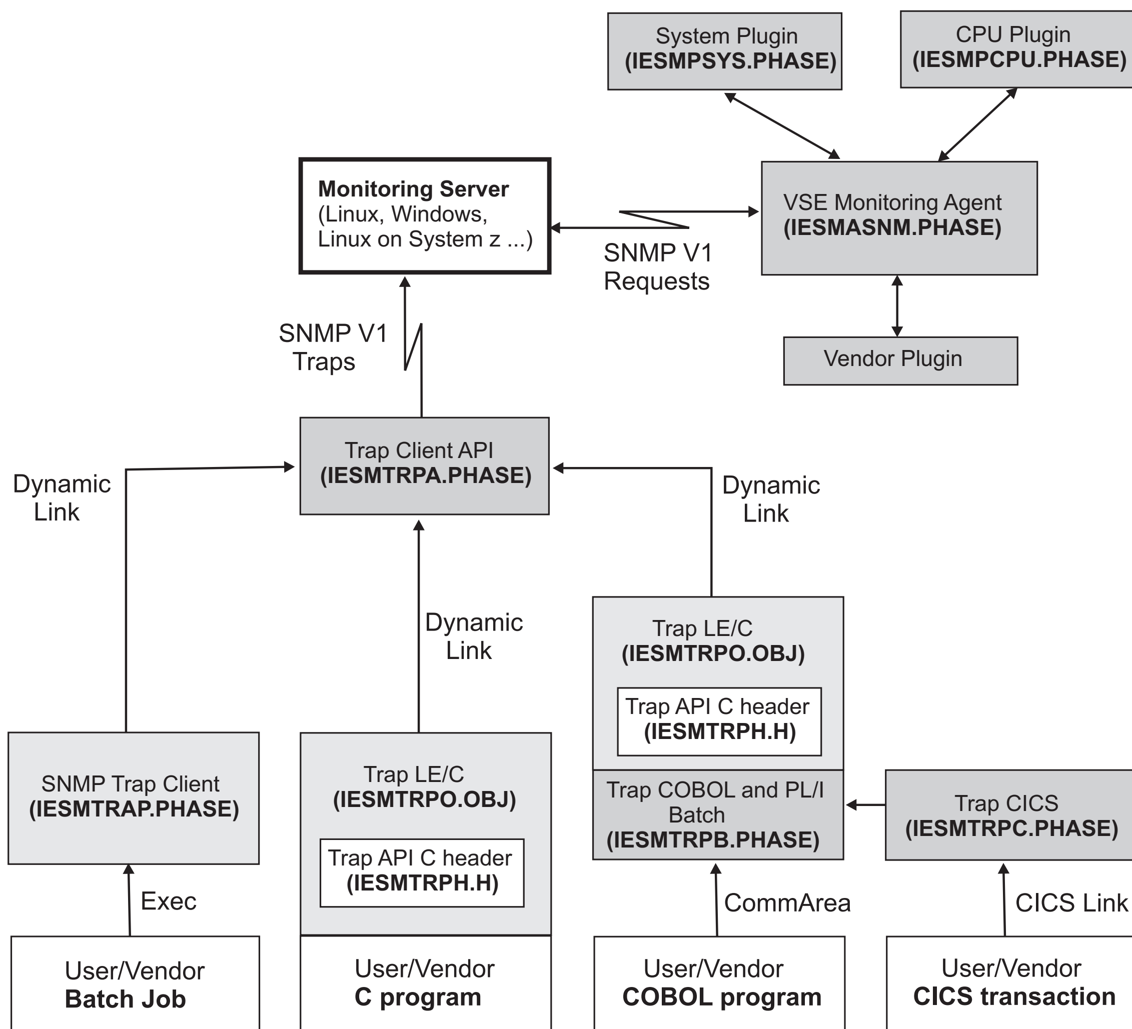


Figure 30. Sending SNMP Traps to the VSE Monitoring Agent

Notes:

1. The Trap Client API (IESMTRPA.PHASE) is available from z/VSE 5.1 onwards.
2. If you have implemented the SNMP Trap Client in any of your batch jobs *before* z/VSE 5.1, you are *not* required to modify these batch jobs. This is because the SNMP Trap Client also uses the Trap Client API without any change to its interfaces or parameters.
3. You can use the Trap Client API (IESMTRPA.PHASE) together with these IBM-supplied interfaces:
   - The Trap COBOL and PL/I Batch interface (IESMTRPB.PHASE)
   - The Trap CICS interface (IESMTRPC.PHASE)

- The Trap LE/C interface (IESMTRPO.OBJ), which includes the Trap API C header (IESMTRPH.H).

4. The members shown in Figure 30 on page 116 (IESMTRAP.PHASE, IESMTRPA.PHASE, IESMTRPB.PHASE, IESMTRPC.PHASE, IESMTRPO.OBJ, and IESMTRPH.H) are provided in sublibrary PRD1.BASE.

Configuring the VSE Monitoring Agent

The VSE Monitoring Agent is a *batch* application that runs in a z/VSE (static or dynamic) partition. It:

- implements an SNMP agent.
- replies to SNMP Version 1 requests from SNMP clients.

Note: For details of the SNMP protocol, refer to the SNMP RFC (RFC 1157).

Figure 31 is a sample job (SKSTMAS) that you can use to start the VSE Monitoring Agent. It is provided in VSE/ICCF Library 59.

```
* $$ JOB JNM=STARTMAS,DISP=L,CLASS=R
// JOB STARTMAS STARTS THE SNMP MONITORING AGENT
* ***************************************************** *
* This Job starts the SNMP MONITORING AGENT.            *
* Please change the ID and the SYSPARM card if necessary *
* ***************************************************** *
// ID USER=VCSRV,PWD=VCSRV
// LIBDEF *,SEARCH=(PRD2.CONFIG,PRD1.BASE,PRD2.SCEEBASE)
// OPTION SYSPARM='00'
// EXEC IESMASNM,PARM='DD:PRD2.CONFIG(IESMASCF.Z)'
/*
/&
* $$ EOJ
```

Figure 31. Job to Start the VSE Monitoring Agent

In Figure 31:

- The VSE Monitoring Agent is started by executing program IESMASNM.
- The `PARM='DD:PRD2.CONFIG(IESMASCF.Z)'` statement specifies where the *configuration file* is located. A sample configuration file (shown in Figure 32) is provided in your PRD2.CONFIG library.

```
* ******************************************** *
* CONFIG FILE FOR z/VSE SNMP MONITORING AGENT   *
* ******************************************** *
* SNMP COMMUNITY NAME:
COMMUNITYNAME = 'public'
* PORT (default SNMP Port 161):
PORT = '161'
* SYSTEM PLUGIN
PLUGIN = 'IESMPSYS'
PARM = 'DD:PRD2.CONFIG(IESMPSCF.Z)'
* CPU PLUGIN
PLUGIN = 'IESMPCPU'
* SAMPLE PLUGIN
* THE SAMPLE PLUGIN IS SHIPED AS SOURCE CODE, YOU
* HAVE TO COMPILE IT, IF YOU WANT TO USE IT
* PLUGIN = 'IESMPSMP'
```

Figure 32. Sample Configuration File Used by the VSE Monitoring Agent

- For security reasons, IBM recommends you change COMMUNITYNAME within the configuration file. To change any settings such as the COMMUNITYNAME, you can use the sample configuration file SKMASCFG (shown in Figure 32 on page 117) and sample job IESMASCFG (shown in Figure 33) to create a new/customized configuration file in PRD2.CONFIG. Sample job IESMASCF is provided in VSE/ICCF Library 59, and automatically includes SKMASCFG.

```
* $$ JOB JNM=IESMASCF,DISP=D,CLASS=0
// JOB IESMASCF CATALOG SNMP MONITORING AGENT CONFIGURATION MEMBER
// EXEC LIBR,PARM='MSHP'
ACCESS S=PRD2.CONFIG
CATALOG IESMASCF.Z REPLACE=Y
* $$ SLI ICCF=(SKMASCFG),LIB=(YY)
/+
/*
/&
* $$ EOJ
```

Figure 33. Job to Replace the Configuration File Used by the VSE Monitoring Agent

Maximizing the Security of SNMP Communication

The VSE Monitoring Agent can only be used with SNMP *Version 1*. The SNMP Version 1 has a *single security mechanism*, which is called the *community name*:

- The community name *must* be included in each SNMP request. This is required by the SNMP protocol.
- If the community name does not equal the community name in your z/VSE system's *configuration file* (shown in Figure 32 on page 117), the VSE Monitoring Agent will ignore the request and will not perform any further processing. (The configuration file usually has the name IESMASCF.Z and is stored per default in library PRD2.CONFIG).

  Note: The community name is **case-sensitive**.
- The default community name in the VSE Monitoring Agent configuration file is `public`. However, this is the default value on many operating systems! Similarly, the community name `private` is also in widespread use. *You are therefore strongly recommended to change the default value*.
- The community name in SNMP therefore has a similar function to a password.
- Note that SNMP requests, responses, and traps, are always sent *unencrypted*. Therefore, if a hacker "listens to" the network traffic, it is relatively easy to obtain the community name.

Using SNMP it is usually also possible to set configuration values on the target systems/servers. However, because of security reasons these SNMP requests are ignored by the VSE Monitoring Agent! Therefore, the client is only allowed to read the data provided by the Monitoring Agent and its plug-ins.

Configuring the IBM-Supplied Monitoring Plug-ins

The VSE Monitoring Agent is designed to be easily extended. IBM has created a *plug-in interface* to monitor:
- z/VSE-specific data.
- Your own data from your own programs.

You can configure the plug-ins that are to be enabled for the VSE Monitoring Agent. These are the currently-available z/VSE monitoring plug-ins:
- The System Plugin, which collects system data.
- The CPU Plugin, which collects CPU data.
- The Sample Plugin, which illustrates how you can write your own plug-in.

The CPU Plug-in and Sample Plug-in are *ready to use*. However, you should change the configuration of the System Plug-in before using it. To change the configuration file of the System Plug-in (shown in Figure 34), use file IESMPSCF (shown in Figure 35) which is also provided in VSE/ICCF library 59. This job automatically includes the configuration file (SKMPSCFG).

```
* ******************************************** *
* CONFIG FILE FOR MONITORING PLUGIN IESMPSYS   *
* ******************************************** *
* ENTER CONTACT INFORMATION AND LOCATION HERE
CONTACT = 'Please change contact in IESMPSCF.Z config
file'
LOCATION = 'Please change location in IESMPSCF.Z config
file'
* THE SYSTEM NAME AND DESCRIPTION ARE OPTIONAL
*DESC = 'z/VSE TEST SYSTEM'
*SYSNAME = 'VSETestSystem'
```

Figure 34. Configuration File Belonging to the System Plug-In

```
* $$ JOB JNM=IESMPSCF,DISP=D,CLASS=0
// JOB IESMPSCF CATALOG MONITORING SYSTEM PLUGIN
CONFIGURATION MEMBER
// EXEC LIBR,PARM='MSHP'
ACCESS S=PRD2.CONFIG
CATALOG IESMPSCF.Z REPLACE=Y
* $$ SLI ICCF=(SKMPSCFG),LIB=(YY)
/*
/&
* $$ EOJ
```

Figure 35. Job to Replace the Configuration File Used by the System Plug-In

- If you want an overview of the data which is handled by the plug-ins, simply view the MIB (Management Information Base) which is provided as IESMPMIB.Z in sublibrary PRD1.BASE. The concept of “MIB” is explained in RFC 2578.
- OIDs with the prefix 1.3.6.1.4.1.2.6.221 are reserved for plug-ins provided by IBM for the z/VSE product.

If you want to create your *own* plugin to gather your own data, the Plugin API is shipped as a C header file (IESMPLGH.H) in library PRD1.BASE together with the C source code of a sample plugin (IESMPSMP.C). For further information, see “Creating Your Own Plug-Ins for the VSE Monitoring Agent” on page 127.

Commands Available With the VSE Monitoring Agent

To enter a command for the VSE Monitoring Agent, enter at the z/VSE Console:

```
msg jobname,data=command
```

where:
- *jobname* is the name of the VSE Monitoring Agent.
- *command* is one of the commands listed in Figure 36.

To obtain a list of all commands available with the VSE Monitoring Agent, you would enter at the z/VSE Console:

```
msg jobname,data=help
```

Figure 36 shows the listing that is then displayed:

```
msg startmas,data=help
AR 0015 1I40I  READY
R1 0045 IESMA105I PLEASE USE "MSG XX,DATA=COMMAND" WITH THE
R1 0045 FOLLOWING COMMANDS:
R1 0045 HELP         DISPLAYS THIS INFORMATION
R1 0045 STATUS       DISPLAYS SERVER STATUS
R1 0045 RESETSTAT    RESETS STATISTICS
R1 0045 LISTOIDS     LISTS HANDLED OIDS
R1 0045 LISTOIDSDET  LISTS HANDLED OIDS (DETAILED)
R1 0045 LISTPLUGINS  LISTS ACTIVE PLUGINS
R1 0045 SHUT         ENDS THE SERVER
R1 0045 SHUTDOWN     ENDS THE SERVER
```

Figure 36. Listing of Commands That Can Be Used With the VSE Monitoring Agent

To obtain the *status* of the VSE Monitoring Agent, you would enter:

```
msg jobname,data=status
```

Figure 37 shows the type of information that is then displayed:

```
msg startmas,data=status
AR 0015 1I40I  READY
R1 0045 IESMA118I AGENT STATUS:
R1 0045 AGENT VERSION:             0004.3000
R1 0045 CONFIG MEMBER:             DD:PRD2.CONFIG(IESMASCF.Z)
R1 0045 PORT:                      161
R1 0045 COMMUNITY STRING:          public
R1 0045 RECEIVED REQUESTS:         5869313
R1 0045 WRONG COMMUNITY STRING:    0
R1 0045 WRONG SNMP VERSION:        0
R1 0045 ANSWERED REQUESTS:         5869313
R1 0045 IESMM002I MONITORING PLUGIN MANAGER STATUS:
R1 0045 MANAGER VERSION:           0004.3000
R1 0045 INSTALLED PLUGINS:         2
R1 0045 HANDLED OIDS:              34
R1 0045 HANDLED OID GROUPS:        1
```

Figure 37. Obtaining the Status of the VSE Monitoring Agent

Examples of How to Collect Data

Using JMibBrowser to Obtain Data: The Open Source program **JMibBrowser** is written in Java. It can use MIBs and send SNMP requests. You can find this program at:

```
http://sourceforge.net/projects/jmibbrowser
```

To use JMibBrowser with z/VSE, you must download the z/VSE MIB (member IESMPMIB.Z in PRD1.BASE) to the `mibs` directory of JMibBrowser. You will then see a new sub-tree in the JMibBrowser MIB tree containing the z/VSE OIDs.

Using SNMP Command-Line Tools to Obtain Data: There are many tools for various operating systems that can send SNMP requests from the command line. They can often perform a “walk”, which requests an SNMP agent for all SNMP agent's OIDs, starting from a specified OID. To use these command line tools, you must usually know which OIDs are supported (because they cannot handle MIBs). To obtain the OIDs, you can either:

- Check the z/VSE MIB (IESMPMIB.Z in PRD1.BASE).
- Enter `msg` *jobname*`,data=LISTOIDS` or `msg` *jobname*`,data=LISTOIDSDET` at the z/VSE Console. For details, see “Commands Available With the VSE Monitoring Agent” on page 120.

Using the SNMP Trap Client in Batch Jobs

As shown in Figure 30 on page 116, you can use the *SNMP Trap Client* in *batch jobs* to send SNMP Version 1 traps to inform one or more monitoring stations/servers when, for example:

- the end of a job stream is reached.
- an error has occurred during a job stream.

The SNMP Trap Client accepts these parameters in batch jobs:

- as EXEC parameters (`EXEC IESMTRAP,PARM='...'`).
- via SYSIPT.

From z/VSE 5.1 onwards, when using the SNMP Trap Client in a batch job you can include *symbolic parameters* in the SYSIPT input.

- The symbolic parameters will be resolved at runtime, and the parameters will be replaced by its values.
- The same rules apply to the symbolic parameters in SYSIPT input as when symbolic parameters are used in JCL.

For details about how to specify symbolic parameters, refer to the chapter “Job Control and Attention Routine” in the *z/VSE System Control Statements*, SC34-2637.

Figure 38 on page 122 is a sample job (SKSTTRAP) that provides an example of how to use the SNMP Trap Client in a batch job. It is provided in VSE/ICCF Library 59 and executes the program **IESMTRAP.PHASE**.

```
* $$ JOB JNM=SNMPTRAP,DISP=L,CLASS=R
// JOB SNMPTRAP STARTS THE VSE SNMP VERSION 1 TRAP CLIENT
* ***************************************************** *
* This Job starts the VSE SNMP VERSION 1 TRAP CLIENT.    *
* Please change the ID and the SYSPARM card if necessary *
* ***************************************************** *
// ID USER=VCSRV,PWD=VCSRV
// LIBDEF *,SEARCH=(PRD2.CONFIG,PRD1.BASE,PRD2.SCEEBASE)
// OPTION SYSPARM='00'
* ***************************************************** *
* This is a sample job, normally you would add the       *
* SNMP TRAP CLIENT at the end of a job stream to         *
* notify you about the results.                          *
* You can add one or more destinations.                  *
* The ADDSYSINF parameter adds system information to     *
* trap packet.                                           *
* If you specify the HELP parameter you will find a      *
* detailed help and a list of all supported parameters   *
* in the job listing.                                    *
* A '*' marks lines as comments                          *
* ***************************************************** *
// EXEC IESMTRAP
DEST=192.168.1.55
DEST=myserver1:162
* DEST=myserver2
OID=1.2.3.4
MSG=This is a test
* MSGINT=99
ADDSYSINF
* This is a comment
* HELP
/*
/&
* $$ EOJ
```

Figure 38. Job to Start the SNMP Trap Client in a Batch Job

To obtain a list of the SNMP Trap Client parameters you can use, specify the Help parameter (`EXEC IESMTRAP,PARM='HELP'`). Figure 39 on page 123 shows how to specify the Help parameter, and the type of information that is then listed.

```
// LIBDEF PHASE,SEARCH=(PRD1.BASE)
// EXEC IESMTRAP,PARM='HELP'
1S54I  PHASE IESMTRAP IS TO BE FETCHED FROM PRD1.BASE
IESMTRAP 2.0.0
IESMA908I IESMTRAP 2.0.0 HELP:
IESMTRAP SENDS A VERSION 1 SNMP TRAP PACKET TO ONE OR MORE DESTINATIONS.
PARAMETERS:
 DEST:      DESTINATION OF THE SNMP TRAP PACKET
            (MULTIPLE ARE POSSIBLE)
            E.G.: DEST=192.168.1.5
                  DEST=192.168.1.5:141
                  DEST=MYSERVER.MYDOMAIN
 OID:       TRAP OID
            E.G.: OID=1.2.3.4.5
 MSG:       MESSAGE
            E.G.: MSG=BACKUP JOB FINISHED
 MSGINT:    MESSAGE (NUMBER)
            E.G.: MSGINT=1234
 COMMUNITY: COMMUNITY STRING (OPTIONAL, DEFAULT: PUBLIC)
            E.G.: COMMUNITY=PRIVATE
 TRAPTYPE:  SPECIFIC CODE (OPTIONAL)
            E.G.: TRAPTYPE=112233
 ADDSYSINF: ADDS SYSTEM INFORMATION TO THE TRAP (OPTIONAL)
ATTENTION: ONLY ONE MSG OR ONE MSGINT IS ALLOWED !
REMARK:
 HELP WILL SHOW THIS PAGE.
 YOU CAN USE * FOR COMMENTS.
 USE - AS CONTINUATION CHARACTER.
1S55I LAST RETURN CODE WAS 0000
EOJ START MAX.RETURN CODE=0000
DATE 02/01/2013, CLOCK 12/49
```

Figure 39. Listing of Parameters That Can Be Used With the SNMP Trap Client

Using the Trap Client API in LE/C Programs

This topic describes how you can use the Trap Client API in your *LE/C programs* to send SNMP Version 1 traps. To do so, you must:

1. Include the Trap API C header (IESMTRPH.H) in your LE/C program.
2. Include the Trap LE/C interface (IESMTRPO.OBJ) during the link-editing of your LE/C program.

The functions you can use in your LE/C programs are described in the Trap API C header (IESMTRPH.H).

Note: Your LE/C programs *can* send traps to multiple destinations at the same time.

The sequence of calls in a LE/C program that uses the Trap Client API would typically be:

1. `TRP_initPrivateData` – initialize the private data structure.
2. `TRP_ addDestination` – add a destination. This function can be called multiple times to add multiple destinations.
3. `TRP_setOid` – set the "sender"-OID.
4. `TRP_setCommunity` – set the community string.
5. `TRP_setMsgInt` or `TRP_setMsg` – add the message (integer or string).
6. `TRP_addSystemInfo` (optional) – add additional system information to the trap.
7. `TRP_checkData` – check that all data is valid and the trap can be sent.
8. `TRP_sendTrap` – send the trap.

9. `TRP_cleanupPrivateData` – cleanup private data (this function also deletes any added destinations).

Functions Provided by the Trap LE/C interface

The Trap LE/C interface (IESMTRPO.OBJ) and Trap API C header (IESMTRPH.H) provide these programming interfaces:

- int TRP_initPrivateData(void** lplpPrivateData);
- int TRP_cleanupPrivateData(void* lpPrivateData);
- int TRP_sendTrap(void* lpPrivateData);
- int TRP_addDestination(destination* dest, void* lpPrivateData);
- void TRP_setDebugMode(int mode, void* lpPrivateData);
- int TRP_setTrapType(int type, void* lpPrivateData);
- int TRP_setMsgInt(int value, void* lpPrivateData);
- int TRP_setMsg(char* value, void* lpPrivateData);
- int TRP_setOid(char* oid, void* lpPrivateData);
- int TRP_setCommunity(char* community, void* lpPrivateData);
- int TRP_addSystemInfo(int addSysInfo, void* lpPrivateData);
- int TRP_checkData(void* lpPrivateData);

For detailed information about these functions, refer to the Trap API C header (IESMTRPH.H). This member is provided in sublibrary PRD1.BASE.

Using the Trap Client API in COBOL and PL/I Programs

This topic describes how you can use the Trap Client API in your *COBOL and PL/I* programs to send SNMP Version 1 traps.

To use the Trap Client API, the Trap COBOL and PL/I Batch interface (IESMTRPB.PHASE) internally uses the Trap LE/C interface (IESMTRPO.OBJ). This is shown in Figure 30 on page 116.

Note: Your COBOL and PL/I programs *cannot* send traps to multiple destinations at the same time. Your COBOL and PL/I programs must instead call the Trap COBOL and PL/I Batch interface (IESMTRPB.PHASE) multiple times.

To use the Trap COBOL and PL/I Batch interface, you should use the Copybook shown in Figure 40 on page 126.

A COBOL program that uses the Trap Client API might look like this:

```
* --------------------------------------------------
*    TRAP INTERFACE Sample Program
* --------------------------------------------------
 Identification Division.
 Program-ID.  TRPINTF.

 Data Division.

 Working-Storage Section.
 01 MTRA-AREA.
    03 AREA-LENGTH              PIC S9(9) BINARY.
    03 DEST                     PIC X(64).
    03 COMMUNITY                PIC X(32).
    03 OID                      PIC X(256).
    03 MSGTYPE                  PIC S9(9) BINARY.
```

```
   03 MSG                        PIC X(256).
   03 MSGSTR REDEFINES MSG       PIC X(256).
   03 MSGINT REDEFINES MSG       PIC S9(9) BINARY.
   03 TRAPTYPE                   PIC S9(9) BINARY.
   03 ADDSYSINF                  PIC S9(9) BINARY.
   03 DEBUG                      PIC S9(9) BINARY.
   03 RET-CODE                   PIC S9(9) BINARY.

01 IESMTRPB         PIC X(8) VALUE 'IESMTRPB'.
Procedure Division.

    Move Length Of MTRA-AREA to AREA-LENGTH.
    Move '9.152.224.43' to DEST.
    Move 0 to RET-CODE.
    Move 'PUBLIC' to COMMUNITY.
    Move '1.2.3.4' to OID.
    Move 0 to DEBUG.
    Move 1 to ADDSYSINF.
    Move 6 to TRAPTYPE.
    Move 1 to MSGTYPE.
    Move 'HELLO VSE WORLD' to MSGSTR.

    DISPLAY "CALLING TRAP INTERFACE ...".
    CALL IESMTRPB USING BY REFERENCE MTRA-AREA.

    DISPLAY "RC:".
    Display RET-CODE.

    Goback.
```

Using the Trap Client API in CICS Programs

This topic describes how you can use the Trap Client API in your *CICS* programs to send SNMP Version 1 traps.

To use the Trap Client API, a *CICS* program (written in C, COBOL, or PL/I) uses the *Trap CICS interface* (IESMTRPC.PHASE). This is shown in Figure 30 on page 116.

1. Your programs call the Trap CICS interface (IESMTRPC) via an `EXEC CICS LINK` command. A COMMAREA is passed that has the same layout as the Trap COBOL and PL/I Batch interface (IESMTRPB.PHASE).
2. The Trap CICS interface (IESMTRPC) internally uses the *Trap COBOL and PL/I Batch interface* (IESMTRPB.PHASE) in order to use the Trap Client API.

The input for the Trap CICS interface COMMAREA should have the same layout as the Copybook for the Trap COBOL and PL/I Batch interface, which is shown in Figure 40 on page 126.

Here is a COBOL example of how the `EXEC CICS LINK` command is used:

```
EXEC CICS LINK PROGRAM('IESMTRPC')
      COMMAREA(MTRA-AREA)
      LENGTH(AREA-LENGTH)
      RESP(RC) RESP2(RC2)END-EXEC
```

Note: Your CICS programs *cannot* send traps to multiple destinations at the same time. Your CICS programs must instead call the Trap CICS interface (IESMTRPC) multiple times.

Copybook Used for the Trap COBOL and PL/I Batch interface

Figure 40 shows the contents of the Copybook that is used as input for the Trap COBOL and PL/I Batch interface (IESMTRPB.PHASE).

```
01 MTRA-AREA.
   03 DESTINATION                  PIC X(64).
   03 COMMUNITY                    PIC X(32).
   03 OID                          PIC X(256).
   03 MSGTYPE                      PIC S9(9) BINARY.
   03 MESSAGE                      PIC X(256).
   03 MSGSTR REDEFINES MESSAGE     PIC X(256).
   03 MSGINT REDEFINES MESSAGE     PIC S9(9) BINARY.
   03 TRAPTYPE                     PIC S9(9) BINARY.
   03 ADDSYSINF                    PIC S9(9) BINARY.
   03 DEBUG                        PIC S9(9) BINARY.
```

Figure 40. Copybook Used as Input for the Trap COBOL and PL/I Batch interface

This is how the fields shown in Figure 40 are used. **Note:** You must supply values in all these fields except for RET-CODE (whose value is returned by the Trap Client API).

AREA-LENGTH
Length of the data-area.

DESTINATION
Destination of the trap. For example, `192.168.1.1` or `192.168.1.2:1234`.

COMMUNITY
Community string of the trap.

OID "Sender"-OID of the trap.

MSGTYPE
Type of the trap:
- String = 1
- Integer = 0

MESSAGE
Message of the trap. Either:
- MSGSTR
- MSGINT

TRAPTYPE
Type of the trap. Default traps have TRAPTYPE = 6.

ADDSYSINF
Add system information to the trap:
- YES = 1
- NO = 0

DEBUG
Enable debug:
- YES = 1
- NO = 0

RET-CODE
Contains the return code sent by the Trap Client API.

Return Codes Generated by the Trap Client API

These return codes might be generated by the Trap Client API:

- TRPERR_NO_ERROR
- TRPERR_MEMORY
- TRPERR_INVALID_PARAMETER
- TRPERR_BER
- TRPERR_ICONV
- TRPERR_SOCKET
- TRPERR_INTERNAL
- TRPERR_SEND_ERROR
- TRPERR_SEND_ERROR_FOR_ALL

For details of these return codes, refer to the Trap API C header (IESMTRPH.H) which is provided in sublibrary PRD1.BASE.

Creating Your Own Plug-Ins for the VSE Monitoring Agent

You might wish to create your own plug-in for the VSE Monitoring Agent so that you can collect your own data. For this purpose, in library PRD1.BASE are provided:

- A plug-in API which is shipped as a C header file (IESMPLGH.H).
- The C source code of a sample plug-in (IESMPSMP.C).

To be able to use the VSE Monitoring Agent, you need to organize your data as OIDs in an OID tree. Please refer to the SNMP and MIB RFCs (RFC 1157 and RFC 2578).

A plug-in consists of a set of callback functions that are called by the VSE Monitoring Agent in a given sequence. Each plug-in consists of a PHASE that you write using LE/VSE-C, which is loaded during VSE Monitoring Agent startup. Your plug-in must provide an IBM-defined interface to allow the VSE Monitoring Agent to call the functions implemented in your plug-in. These callback functions are now described:

1. By loading the plug-in phase (CDLOAD), the VSE Monitoring Agent obtains the entry point of the plug-in (`PluginMainEntryPoint`).
2. The entry point of the plug-in is called multiple times to obtain all entry points of the other plug-in functions. Each call returns the entry point of a function specified by the parameter `iFuncID`.
3. When the previous step is completed, the agent has a list of the entry points in the plug-in phase.

This is the sequence after the plug-in has been loaded:

1. The VSE Monitoring Agent calls the `MPL_Init` function. This function is used to initialize all request-independent resources (allocate storage, open database connections, and so on).
2. The VSE Monitoring Agent calls the `MPL_GetCountOfHandledOIDs` and `MPL_GetHandledOID` functions immediately after calling the `MPL_Init` function.
   a. `MPL_GetCountOfHandledOIDs` returns the count of handled OIDs that are accepted by the plug-in.
   b. `MPL_GetHandledOID` is called for every handled OID.

3. The VSE Monitoring Agent can now use the above information in order to direct a request to a specific plug-in.

Therefore, OIDs must be unique across all plug-ins. To find the right place in the OID tree, you can first use the OID prefix 1.3.6.1.3. for testing purposes.

Later you should use either:
- Your own company's prefix.
- Already-defined standard OIDs. For example, if you want to provide network data then you can use the network OIDs described in RFC 1213.

Refer to the header file IESMPLGH.H file for more details.

Note: OIDs with the prefix 1.3.6.1.4.1.2.6.221 are reserved for plug-ins provided by IBM for the z/VSE product.

Chapter 14. Using GDPS Support for High Availability

GDPS is an abbreviation for *Geographically Dispersed Parallel Sysplex*® and can be used for high availability and disaster recovery in large computer environments. This chapter describes how you can use the *GDPS Client* to enable availability data to be collected from your z/VSE systems and sent to a central *GDPS K-System* running under z/OS.

It contains these main topics:

- "Overview of How the GDPS Client Is Used"
- "Configuring the GDPS Client" on page 130
- "Starting the GDPS Client" on page 131
- "Communication Between GDPS Client and GDPS K-System" on page 131
- "Commands Available With the GDPS Client" on page 132

Overview of How the GDPS Client Is Used

The IBM *GDPS K-System*, which runs under z/OS, is responsible for monitoring and managing all connected systems. To monitor and manage the different systems, a protocol is used that enables connections to be established between the GDPS K-System and its connected systems. z/VSE's *GDPS Client* provides a *client* connection to a GDPS K-System which consists of:

- A command receiver (`cmdreceiver`)
- An event sender.

z/VSE's support for GDPS is shown in Figure 41.

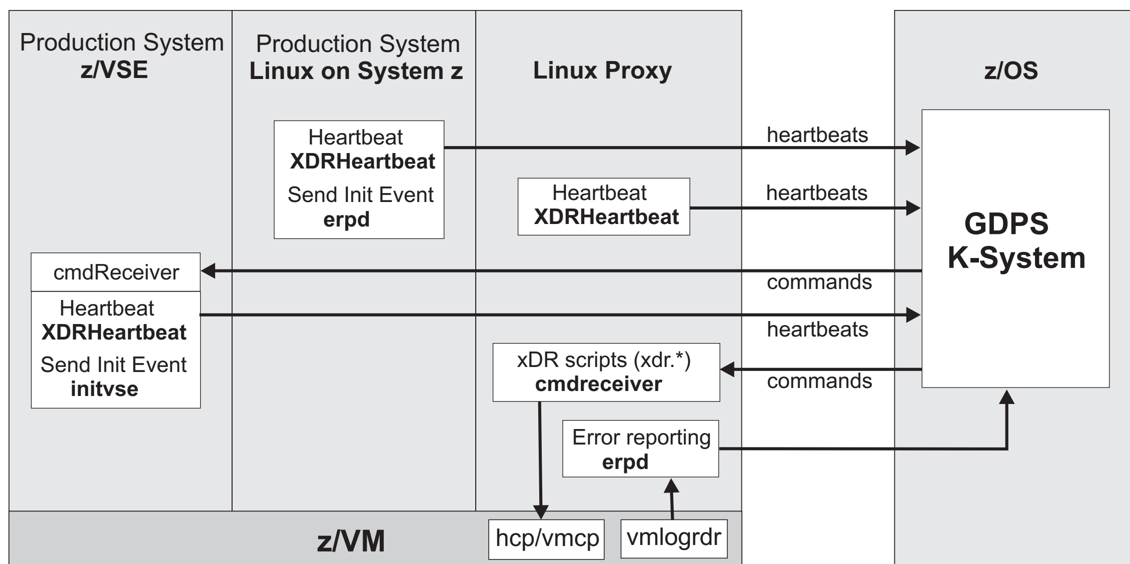


Figure 41. Overview of z/VSE's GDPS Support

Notes:

1. A GDPS K-System *cannot* be used with a z/VSE system that runs in a LPAR. It can only be used with a z/VSE system that runs *under z/VM*.

© Copyright IBM Corp. 2000, 2013

2. z/VSE only supports the heartbeat mechanism of GDPS.
3. A GDPS K-System can only *monitor* (and not manage) a z/VSE system.
4. A GDPS K-System can manage the z/VM under which a z/VSE system is running. Therefore, a GDPS K-System can *indirectly manage* a z/VSE system (for example, by initiating a disk swap or shutting down a z/VM guest where a z/VSE system is running).

Configuring the GDPS Client

Before the GDPS Client can be started, you must configure the GDPS Client using sample Job SKGDPSCF (provided in VSE/ICCF library 59).

```
$$ JOB JNM=GDPSCFG,DISP=D,CLASS=0
// JOB GDPSCFG CATALOG GDPS CLIENT CONFIGURATION MEMBER
// EXEC LIBR,PARM='MSHP'
ACCESS S=PRD2.CONFIG
CATALOG IESGDPCF.Z REPLACE=Y
*-------------------------------------------*
* GDPS CLIENT CONFIGURATION *
*-------------------------------------------*
* NAME OF THIS GDPS CLIENT (MUST BE SAME AS YOUR HOSTNAME)
GDPSCNAME='MYVSE'
* RECEIVER (LOCAL) PORT
RECEIVERPORT='16111'
* INTERVAL (IN SECONDS) BETWEEN HEARTBEATS
HEARTBEATINTERVAL='10'
* TIMEOUT
TIMEOUT='60'
* CURRENT MAINTENANCE STATUS
MAINTENANCE='OFF'
* CURRENTLY ACTIVE SITE
SITE='1'
*-----------------------*
* SITE 1 CONFIGURATION *
*-----------------------*
* IP/HOSTNAME OF THE GDPS K-SYSTEM
SITE1_KSYSTEM='123.123.1.1'
* PORT OF THE GDPS K-SYSTEM
SITE1_KSYSTEMPORT='16112'
*-----------------------*
* SITE 2 CONFIGURATION *
*-----------------------*
* IP/HOSTNAME OF THE GDPS K-SYSTEM
SITE2_KSYSTEM='123.123.1.2'
* PORT OF THE GDPS K-SYSTEM
SITE2_KSYSTEMPORT='16112'
/+
/*
/&
$$ EOJ
```

Figure 42. Configuring the GDPS Client Using Job SKGDPSCF

Per default, the job shown in Figure 42 creates a member with name *IESGDPCF.Z* in sublibrary *PRD2.CONFIG*. If required, change these default values before submitting this job.

You can also specify your own values for these parameters in the job shown in Figure 42:

GDPSCNAME
: The name/ID that will be displayed in the GDPS K-System. **Note:** GDPSCNAME must be the same as the hostname of your z/VSE system!

RECEIVERPORT
The local port to be used for incoming requests.

HEARTBEATINTERVAL
The time interval (in seconds) between which heartbeats are sent to the GDPS K-System.

TIMEOUT
The timeout (in seconds) during which the GDPS K-System will wait for the heartbeat of this GDPS Client.

MAINTENANCE
Displays whether or not the GDPS Client starts in *maintenance mode*.

SITE The currently active site.

IP/Hostname
The IP/hostname of each of the two sites used with the GDPS K-System.

Port The port name of each of the two sites used with the GDPS K-System.

Starting the GDPS Client

The GDPS Client is a *batch* application that runs in a z/VSE (static or dynamic) partition.

Figure 43 is a sample job (SKSTGDPS) that you can use to start the GDPS Client. It is provided in VSE/ICCF Library 59. This job executes phase IESGDPSC.PHASE stored in sublibrary PRD1.BASE.

```
$$ JOB JNM=STRTGDPS,DISP=L,CLASS=R
// JOB STRTGDPS STARTS THE GDPS CLIENT
* ***************************************************** *
* This Job starts the GDPS CLIENT.                      *
* Please change the ID and the SYSPARM card if necessary *
* ***************************************************** *
// ID USER=VCSRV,PWD=VCSRV
// LIBDEF *,SEARCH=(PRD2.CONFIG,PRD1.BASE,PRD2.SCEEBASE)
// OPTION SYSPARM='00'
// EXEC IESGDPSC,PARM='DD:PRD2.CONFIG(IESGDPCF.Z)'
/*
/&
$$ EOJ
```

Figure 43. Sample Job SKSTGDPS to Start the GDPS Client

Communication Between GDPS Client and GDPS K-System

This topic provides an overview of how the GDPS Client communicates with the GDPS K-System.

1. If the GDPS Client *was* in maintenance mode the last time it was shut down, it will also *start* in maintenance mode. Therefore, a STOPMAINTENANCE command must be issued in order to start heartbeating. If the GDPS Client is not in maintenance mode, after being started it attempts to send an `init` packet to the configured GDPS K-System.
2. If the GDPS Client *is* in maintenance mode, no communication occurs until a command is entered to end the maintenance mode (for example, the STOPMAINTENANCE command which is described in "Commands Available With the GDPS Client" on page 132).
3. The reply that the GDPS K-System returns to the `init` packet might include a command that forces the GDPS Client to switch to the second site. After the

GDPS Client has processed the reply, it starts to send heartbeats to the GDPS-K system in the defined *interval* (see HEARTBEATINTERVAL in Figure 42 on page 130).
4. If the GDPS K-System sends a switch command to the GDPS Client, the GDPS Client will switch the site and start sending heartbeats to the second GDPS K-System.
5. During a controlled shutdown of the GDPS Client, the GDPS Client will first send a "maintenance" packet to the GDPS K-System, to ensure that the GDPS K-System will no longer wait for heartbeats.

To switch the GDPS Client to maintenance mode, you must use the STARTMAINTENANCE command. To end the maintenance mode, you must use the STOPMAINTENANCE command. Both of these commands are described in "Commands Available With the GDPS Client."

Notes:
1. The GDPS Client will update the configuration member if:
   - The GDPS Client has to change the site or the maintenance mode.
   - The user resets values within the configuration file via parameter changes.
2. To ensure that important changes are not lost, the GDPS Client will automatically create a `.SAVE` backup file in the same library.

Commands Available With the GDPS Client

To enter a command for the GDPS Client, enter at the z/VSE Console:

```
msg jobname,data=command
```

where:
- *jobname* is the name of the GDPS Client.
- *command* is one of the commands listed below.

These are the commands you can enter at the z/VSE Console:

HELP Displays an overview of the valid commands and a short description of each command.

STATUS
Displays the status of the GDPS Client, including the currently active site, how many requests were received and how many heartbeats were sent

RESETSTAT
Resets the counter for the received requests and the sent heartbeats.

STARTMAINTENANCE
Starts the maintenance mode.

STOPMAINTENANCE
Stops the maintenance mode.

SETGDPSCNAME=*yyy*
Sets the name/id of the GDPS Client (which you will see at the GDPS K-System) to the value of *yyy*.

SETSITE1KSYSTEM=*yyy*
Sets the IP/hostname of the GDPS K-System for site 1 to the value of *yyy*.

SETSITE1KSYSTEMPORT=*x*
Sets the port of the GDPS K-System for sites 1 to *x*.

SETSITE2KSYSTEM=*yyy*
Sets the IP/hostname of the GDPS K-System for site 2 to the value of *yyy*.

SETSITE2KSYSTEMPORT=*x*
Sets the port of the GDPS K-System for sites 2 to *x*.

SWITCHSITE
Switches the site. For example, if currently site 1 is active, then it will switch to site 2. **Note:** If the GDPS Client is initialized at site 1 and you switch with this command to site 2, the GDPS K-System of site 1 may interpret this as an error.

SETRECEIVERPORT=*x*
Sets the (local) port for incoming requests from the GDPS K-System. **Note:** This port is not configurable at the GDPS K-System, because the GDPS Client sends this port in the `init` packet to the GDPS K-System.

SETINTERVAL=*x*
Sets the interval in seconds, when the GDPS Client sends heartbeats.

SETTIMEOUT=*x*
Sets the timeout in seconds. The GDPS K-System will wait for this timeout for heartbeats, until it tries to recover the system.

RELOADCONFIG
Reloads the configuration member.

FORCESHUT
Shuts the GDPS Client down, but does not send a `maintenance` packet to the GDPS K-System. **Note:** If the GDPS Client is initialized and you shut down using this command, the GDPS K-System will interpret this as an error!

SHUT Shuts the GDPS Client down, but sends a `maintenance` packet to the GDPS K-System before.

SHUTDOWN
Same as shut.

Part 4. PROGRAMMING

Chapter 15. Using VSE Java Beans to Implement Java Programs 141
Where VSE Java Beans Are Installed and Used . . 142
How JavaBeans and EJBs Compare to VSE Java Beans 142
Contents of the VSE Java Beans Class Library . . 143
Example of a Javadoc for a VSE Java Bean . . . 145
Using the Callback Mechanism of VSE Java Beans 146
Example of Using VSE Java Beans to Connect to the Host 148
Step 1: Create a VSEConnectionSpec. 149
Step 2: Create a VSESystem 149
Example of Using VSE Java Beans to Connect to the Host via SSL 149
Step 1: Prompt for IP Address, User ID, Password. 150
Step 2: Create a Connection Specification for the VSE System 150
Specifying the SSL Properties: Alternative 1 . . 150
Specifying the SSL Properties: Alternative 2 . . 151
Specifying the SSL Properties: Alternative 3 . . 151
Main Method of the Class Used in This Example 152
Implementation of the ConfirmCertificate Method 152
Example of Using VSE Java Beans to Submit Jobs to the Host 152
Step 1: Prompt for IP Address, User ID, Password. 153
Step 2: Create a Connection Specification for the VSE System 153
Step 3: Submit a Job File. 153
Step 4: Create the Job File and Send It to the Host 153
Example of Using VSE Java Beans to Access the Operator Console 155
Step 1: Prompt for IP Address, User ID, Password. 155
Step 2: Create a Connection Specification for the VSE System 155
Step 3: Create a Console Instance and Send a Command 156
Step 4: Obtain Messages One Line at-a-Time . . 156
Example of Using VSE Java Beans to Access VSAM Data 156
Step 1: Define the Local Variables 157
Step 2: Prompt for IP Address of VSE System, User ID, Password 157
Step 3: Create Connection Specification for the VSE System 157
Step 4: Create a VSEResourceListener 157
Step 5: Get VSAM Records from the z/VSE Host 158
Step 6: Display VSAM Records 158
Step 7: Insert a VSAM Record in the VSAM Cluster 158
Step 8: Prompt the User to Enter Column Values 159
Example of Using VSE Java Beans to Access DL/I Data 159
Step 1: Create a VSE System Instance and Get Access to DL/I 160
Step 2: Schedule the PSB 160
Step 3: Get a PCB 160
Step 4: List DL/I Segments. 161
Step 5: Insert or Update a DL/I Segment . . . 161
Step 6: Delete a DL/I Segment 163
Step 7: Terminate the PSB 163
Example of Using VSE Java Beans to Access VSE/POWER Data 163
Step 1: Prompt for IP address, User ID, and Password. 164
Step 2: Create a Connection Specification for the VSE System 164
Step 3: Create a VSEResourceListener 164
Step 4: Scan Compile Outputs for Errors . . . 165
Example of Using VSE Java Beans to Access Librarian Data 165
Step 1: Prompt for IP address, User ID, and Password. 165
Step 2: Create a Connection Specification for the VSE System 166
Step 3: Create a VSEResourceListener 166
Step 4: Obtain a List of Libraries From the VSEResourceListener 166
Step 5: Obtain and Count a List of Sub-Libraries 167
Step 6: Obtain the Instance of the PRD2.CONFIG Sub-Library 167
Step 7: Obtain a List of Members in the PRD2.CONFIG Sub-Library 167
Step 8: Obtain Properties of the First Member 167
Step 9: Download the Member to a Local Disk 168
Example of Using VSE Java Beans to Access VSE/ICCF Data 168
Step 1: Prompt for IP address, User ID, and Password. 168
Step 2: Create a Connection Specification for the VSE System 168
Step 3: Create a VSEResourceListener 169
Step 4: Obtain a List of ICCF Libraries From the VSEResourceListener 169
Step 5: Download a Specific VSE/ICCF Member 170
Step 6: Download a Specific VSE/ICCF Member (Very Fast Method) 170
Using the VSE Navigator Application 170
Prerequisite for Using the VSE Navigator . . . 172
Migrating From Earlier Versions 172
Installing the VSE Navigator 173
Starting the VSE Navigator Client 173
Adding Your Own VSE Navigator Plug-Ins . . 174
Using the VSE Health Checker Application . . . 175
Prerequisite for Using the VSE Health Checker 176
Migrating From Earlier Versions 176

© Copyright IBM Corp. 2000, 2013

Installing the VSE Health Checker 177
Starting the VSE Health Checker Client. . . . 177

Chapter 16. Using JDBC to Access VSAM Data 179
SQL Statements That Are Supported by JDBC . . 179
Relational and VSE Java Beans Terminology . . . 182
Specifying Table Names 182
Example of Using JDBC to Access VSAM Data . . 182
Step 1. Define the Local Variables 183
Step 2: Prompt for IP address, User ID, and Password. 183
Step 3. Establish a Connection to the z/VSE Host 183
Step 4. Display a List of Rows in the Database 184
Step 5. Process Result-Set Returned From JDBC 184
Step 6. Add a New Record 185

Chapter 17. Using Java Applets to Access Data 187
How Applets Are Used in 2-Tier Environments . . 187
How Applets Are Used in 3-Tier Environments . . 189
How the VSEAppletServer Is Used 191
Disadvantages and Restrictions Of Using Applets 191
Running the Sample Data-Mapping Applet . . . 192
Description of the Data-Mapping Applet . . . 192
Activities Required on the z/VSE Host 194
Deploying the Data-Mapping Applet 194
Calling the Data-Mapping Applet 195
How Various Web Browsers Search for JAR and Class Files 195
Setting Up the Data-Mapping Applet Class . . 195
Initializing the Data-Mapping Applet 196
Re-Displaying or Leaving an HTML Page . . . 196
Using the Data-Mapping Applet to Add a Map to a VSAM Cluster 196
Using the Data-Mapping Applet to Modify a Map 197
Using the Data-Mapping Applet to Modify a Map's Data Fields 198
Running the Data-Mapping Applet Locally Using the AppletViewer 200
Running the Sample VSAM Applet 201
Description of the VSAM Applet 201
Getting Started With the Sample VSAM Applet 203
1. Create an HTML File to Call the VSAM applet 203
2. Compile VSAMSEL.C 203
3. Define VSAMSEL to the DB2 Server for VSE 204
4. Define the VSAM Data Cluster. 204
5. Create the JAR File for the VSAM applet 206
Calling the VSAM Applet 206
Description of DB2ConnectorJDBCApplet.java (the Client-Side Program) 208
Step 1. Import the JDBC (Java Database Connectivity) Classes. 208
Step 2. Load the Required JDBC Driver Class 209
Step 3. Implement the init() Method. 209
Step 4. Establish the Connection to z/VSE Database via DB2 Connect 209
Step 5. Call VSAMSEL 210
Description of VSAMSEL 210
Step 1. Include Header File iesvsql.h in VSAMSEL 211
Step 2. Initialize the VSAMSQL CLI Environment 211
Step 3. Initiate the Read of the VSAM Records 212
Step 4. Obtain the Results of the Query Statement 212
Step 5. Deallocate the VSAMSQL CLI Environment 213
Step 6. Assign Local Output Variables to Host Output Variables 214
Running the Sample DL/I Applet 214
Description of the DL/I Applet 214
Getting Started With the Sample DL/I Applet 216
1. Create an HTML File to Call the DL/I applet 216
2. Compile DLIREAD.C 217
3. Define DLIREAD to the DB2 Server for VSE 217
4. Define the DL/I Database 217
5. Create the JAR File for the DL/I applet 217
Calling the DL/I Applet. 218
Description of DB2DLIConnectorJDBCApplet.java (the Client-Side Program) 220
Step 1. Prepare the SQL Statement to Call DLIREAD 220
Step 2. Call DLIREAD 221
Step 3. Check the Return Code from DLIREAD 221
Step 4. Reset the Connection to the sqlds Database 222
Description of DLIREAD 222
Step 1. Define Variables for AIBTDLI, and I/O Area 223
Step 2. Define the Parameters for DLIREAD 223
Step 3. Define DLIREAD's Parameters to COBOL 224
Step 4. Obtain the Results of the Query Statement 224
Step 5. Check for Further DL/I Segments . . 226
Step 6. Run the Error-Handling Routines . . 226

Chapter 18. Using Java Servlets to Access Data 227
How Servlets Are Used in 3-Tier Environments . . 227
Compiling and Calling Servlets 229
How the WebSphere Application Server Stores Session Information 230
Example of How to Implement a Servlet 230
General Description of the Sample Servlet . . . 231
Creating the VSAM Clusters for the Sample . . 231
HTML Constructs Used With the Sample . . . 232
How a Servlet Can Create Tables in HTML 232
Using Forms to Obtain a User's Input . . . 232
Sample Servlet Step 1: Display a List of Flights. 234
Sample Servlet Step 2: Get Flight Instances from the Host 235
Sample Servlet Step 3: Display the Properties of a Flight 236

Sample Servlet Step 4: Place an Order . . . 239
Sample Servlet Step 5: Create a New Flight 241
Sample Servlet Step 6: Create a New Order 242

Chapter 19. Using Java Server Pages to Access Data 245
How JSPs Are Used in 3-Tier Environments . . . 245
Example of a Simple Java Server Page 247

Chapter 20. Using EJBs to Represent Data . . 249
Overview of the EJB Architecture. 249
Overview of How EJB Containers are Used . . 250
How EJBs Compare to JavaBeans / Java Servlets 251
Implementing Your Client Applications. 252
How an EJB Client Accesses EJBs. 253
Example of Using EJBs to Access VSAM Data . . 254
Example of Implementing VSAM-Based EJBs. . . 256
Step 1: Define the Sample's VSAM Cluster . . 257
Step 2: Create the Record Layout for Employees 257
Step 3: Specify the EJB's Home Interface . . . 258
Step 4: Specify the EJB's Remote Interface . . . 258
Step 5: Implement the RecordPK Class 259
Step 6: Implement the EJB Code 259
Step 6.1: Implement the Methods of the EntityBean Interface 260
Step 6.2: Access z/VSE Host and Get Records from the Database. 262
Step 6.3: Implement the Set & Get Methods to Access the Data Fields 263
Step 7: Compile the Java Source Files 264
Step 8: Deploy the EJBs 264
Step 9: Access the EJBs from an EJB Client . . 265
Prerequisites for Accessing the EJBs from an EJB Client 265
How EJBs Are Accessed from an EJB Client 265
Sample EJB Client Source Code for Accessing EJBs from an EJB Client 266

Chapter 21. Extending the Java-Based Connector 269
Implementing a Server Plugin 269
Implementing a PluginMainEntryPoint Function 274
Implementing a SetupPlugin Function 275
Implementing a CleanupPlugin Function . . . 276
Implementing a GetHandledCommands Function 277
Implementing a SetupHandler Function . . . 278
Implementing an ExecuteHandler Function . . 279
Implementing a CleanupHandler Function . . 281
Creating Your Own Plugin Callback Functions 282
Action Codes Supported by the VSE Connector Server 282
Utility Functions Supported by the VSE Connector Server 284
Using the IBM-Supplied Server Plugin Example 285
Registering and Compiling Your Server Plugin 285
Implementing a Client Plugin 286
Using the VSEPlugin class 286
General Considerations When Designing Your Plugin 288
Specifying the Protocol Between VSE Connector Server and Plugin 288
Choosing the Access Method to the Data / Application 289
Considerations for ASCII / EBCDIC and Big / Little Endian 289
Deciding Which Requests / Functions Should Be Supported 290
Transferring Data Over the Network 290
Structuring the Client Plugin's View 290

Chapter 22. Using the Database Call Level Interface to Access Data 291
DBCLI Programming Concepts 291
Initializing and Terminating the API Environment 291
Connecting and Disconnecting to/from the DBCLI Server and Vendor Database 292
Logical Units of Work (Transactions) 292
Executing SQL Statements 293
Cursors 294
Database Meta Data 295
Connection Pooling 295
Programming Restrictions and Requirements . . . 296
Using the DBCLI in COBOL 296
Using The DBCLI in PL/I 297
Using The DBCLI in C 297
Using the DBCLI in Assembler 298
Using the DBCLI in REXX 299
Syntax and Parameters of a DBCLI Function Call 299
DBCLI Functions (Reference Information) 300
Roadmap of Where the DBCLI Functions Are Used 300
BINDCOLUMN 301
BINDPARAMETER 304
CLOSECURSOR 307
CLOSESTATEMENT 308
COMMIT. 308
CONNECT 309
CONNECTSSL 311
DBATTRIBUTES 313
DBBESTROWIDENT 315
DBCATALOGS 317
DBCOLUMNPRIV. 318
DBCOLUMNS 320
DBCROSSREFERENCE 322
DBEXPORTEDKEYS 325
DBIMPORTEDKEYS 327
DBINDEXINFO 329
DBPRIMARYKEYS 332
DBPROCEDURECOLS 333
DBPROCEDURES 335
DBSCHEMAS 337
DBSUPERTABLES 338
DBSUPERTYPES 339
DBTABLEPRIV 341
DBTABLES 343
DBTABLETYPES 345
DBTYPEINFO 346
DBUDTS 347
DBVERSIONCOLS 349

DISCONNECT 350
EXECUTE 351
FETCH 352
GETCOLUMNINFO 353
GETCONNATTR 355
GETENVATTR 369
GETLASTERROR 371
GETMORERESULTS 372
GETNUMCOLUMNS. 373
GETNUMPARAMETERS 374
GETPARAMETERINFO 374
GETROWNUMBER 376
GETSTMTATTR 376
GETUPDATECOUNT 378
INITENV 379
INITSSL 380
PREPARECALL 381
PREPARESTATEMENT 382
RELEASESAVEPOINT 384
ROLLBACK 385
SETCONNATTR 385
SETENVATTR 386
SETPOS 387
SETSAVEPOINT 388
SETSTMTATTR. 389
TERMENV 389
Performance Considerations When Using the DBCLI. 390
Using SSL 390
Pre-fetching 390
Binding Columns 391
Investigating the Cause of Errors When Using the DBCLI. 391
Return Codes Used by the DBCLI 392

Chapter 23. Using the DB2-Based Connector to Access Data 393
How You Use DB2 Stored Procedures 393
Grouping Stored Procedure Servers 394
Programming Requirements When Using DB2 Stored Procedures 394
Using DB2 Stored Procedures to Access VSAM Data 395
Overview: Accessing VSAM Data via DB2 Stored Procedures 395
Using the Call Level Interface: Activities on the Requestor 397
Using Call Level Interface: Activities on the z/VSE host 398
Example of the Syntax of a CLI Function – VSAMSQLCloseTable. 399
VSAMSQLCloseTable - Close a specified table (Cluster) 399
Program Flow When Using the VSAMSQL Call Level Interface 399
SQL Statements Supported by VSAMSQL Call Level Interface 400
Using DB2 Stored Procedures to Access DL/I Data 402
Overview of the AIBTDLI Interface 403
Creating Programs That Use AIBTDLI 405
Invoking the AIBTDLI Interface 406
Format of the Scheduling Call 406
Format of the Database Call 407
Format of the Termination Call 407
Format of the Roll Back Call 407
Compiling and Link-Editing Your Programs . . 408
Return and Status Codes 408
Format of the AIB – User Section 408
How Return Code X'FF00' Is Used 409
Errors That Do Not Produce a Return Code 409
Scheduling with Single and Multiple MPS Systems 409
Task Termination and Abend Handling 410
Messages and Return Codes 411

Chapter 24. Using SOAP for Inter-Program Communication 413
Overview of z/VSE Support for Web Services and SOAP 414
Overview of the SOAP Syntax. 414
Overview of Web Service (SOAP) Security. . . . 415
Comparison of Transport-Layer Security and Message-Layer Security 415
Using Authentication With Web Service Security 416
How the z/VSE Host Can Act As the SOAP Server 418
Using Web Service Security Features When z/VSE Acts As the SOAP Server 419
How the z/VSE Host Can Act As the SOAP Client 420
Using Web Service Security Features When z/VSE Acts As the SOAP Client 421
How the IBM-Supplied SOAP Control Blocks Are Used 421
How the SOAP_PARAM_HDR Control Block Is Used 422
How the SOAP_PROG_PARAM Control Block Is Used 424
How the SOAP_DEC_PARAM Control Block Is Used 425
Configuring the z/VSE SOAP Engine 426
Mapping Long-Names to Short-Names 428
Description of the IBM-Supplied SOAP Service (getquote.c) 428
Description of the IBM-Supplied SOAP Client (soapclnt.c) 430
Using a Java SOAP Client 432
Running the IBM-Supplied SOAP Sample 433
Step 1: Download and Install the Java SOAP Client Packages on the Client 433
Step 2: Extract and Install the Required Java Programs. 433
Step 2.1: Create a New Directory 433
Step 2.2: Extract JAR Files and Place in Your Directory 433
Step 3: Compile /Link the Sample C Programs, and Define Them to CICS 434
Step 4: Configure CICS to Use CICS Web Support 435
Step 5: Define the SOAP Server to CICS . . . 435
Step 5: Activate the ASCII to EBCDIC Converter 435
Step 6: Compile the Java Sample 435
Step 7: Run the Java SOAP Client Sample . . . 436
Step 8: Run the C-Program SOAP Client Sample 436

Writing Your Own SOAP Programs 436

Chapter 25. Using the VSE Script Connector for Non-Java Access 437
How the VSE Script Connector Can Be Used . . . 437
Overview of the Protocol Used Between Client and Server 439
Using SSL-Encrypted Connections 440
Writing VSE Scripts Using the VSE Script Language. 440
General Rules That Apply to the VSE Script Language. 440
VSE Script Language Built-In General Functions 441
VSE Script Language: Built-In String Functions 442
VSE Script Language: Built-In Console Functions. 443
VSE Script Language: Built-In POWER Functions. 443
VSE Script Language: Librarian Functions . . . 443
VSE Script Language: Built-In VSAM Functions 444
Sample Files You Can Use for Writing VSE Script Clients 445
Example of Writing a VSE Script Client (and Its VSE Script) 445
Step 1: Setup the VSE Script Server Properties File. 446
Step 2: Setup the Connections Properties File 446
Step 3: Define the Sample VSAM Data 446
Step 4: Modify the Sample VSE Script 447
Step 5: Start the VSE Connector Server on the z/VSE Host 448
Step 6: Start the VSE Script Server Locally. . . 449
Step 7(a): Open the Sample Lotus 1-2-3 Spreadsheet File 449
How the Sample VSE Script is Defined in Lotus 1-2-3 450
Step 7(b): Open the Sample MS Office Spreadsheet 452
How the Sample VSE Script is Defined in MS Office 453
Step 7(c): Start a Sample VSE Script from the Command Line. 455
Obtaining Data From the Middle-Tier Using VSE Script Clients 455
Using the VSE Script Client That Runs in Batch 457
Using the VSE Script Client That Runs Under CICS 459

Chapter 15. Using VSE Java Beans to Implement Java Programs

This chapter describes how you use the VSE Connector Client's *VSE Java Beans* within 2-tier and 3-tier environments.

It contains these main topics:

- "Where VSE Java Beans Are Installed and Used" on page 142
- "How JavaBeans and EJBs Compare to VSE Java Beans" on page 142
- "Contents of the VSE Java Beans Class Library" on page 143
- "Example of a Javadoc for a VSE Java Bean" on page 145
- "Using the Callback Mechanism of VSE Java Beans" on page 146
- "Example of Using VSE Java Beans to Connect to the Host" on page 148
- "Example of Using VSE Java Beans to Connect to the Host via SSL" on page 149
- "Example of Using VSE Java Beans to Submit Jobs to the Host" on page 152
- "Example of Using VSE Java Beans to Access the Operator Console" on page 155
- "Example of Using VSE Java Beans to Access VSAM Data" on page 156
- "Example of Using VSE Java Beans to Access DL/I Data" on page 159
- "Example of Using VSE Java Beans to Access VSE/POWER Data" on page 163
- "Example of Using VSE Java Beans to Access Librarian Data" on page 165
- "Example of Using VSE Java Beans to Access VSE/ICCF Data" on page 168
- "Using the VSE Navigator Application" on page 170
- "Using the VSE Health Checker Application" on page 175

© Copyright IBM Corp. 2000, 2013

Where VSE Java Beans Are Installed and Used

All VSE Java Beans classes are contained in one Java archive **VSEConnector.jar**, which are included in the VSE Connector Client.

You will usually install the VSE Java Beans in this way:

1. Install the VSE Connector Client on the workstations where you develop your Java applications.
2. When these Java applications are ready to be implemented in your production systems, copy these files to either the Web clients of 2-tier environments, or physical/logical middle-tier of 3-tier environments:
   - VSEConnector.jar
   - cci.jar
   - ibmjsse.jar
   - ibmpkcs.jar

The use of VSE Java Beans is described for:

- applets in 2-tier environments, in Figure 102 on page 188.
- applets in 3-tier environments, in Figure 103 on page 190.
- servlets in 3-tier environments, in Figure 117 on page 228.
- JSPs in 3-tier environments, in Figure 131 on page 246.
- EJBs in 3-tier environments, in Figure 136 on page 255.

How JavaBeans and EJBs Compare to VSE Java Beans

You should distinguish between *JavaBeans*, *Enterprise Java Beans (EJBs)*, and *VSE Java Beans*:

- *JavaBeans* are usually *visual* components, such as push buttons, sliders, and list boxes. *VisualAge® for Java* allows you to assemble dialogs using these components without any programming effort. For more information, refer to:

  `http://www.javasoft.com/beans/docs`
- Enterprise Java Beans (EJBs) are *distributed* Java Beans. The term *distributed* is used because:
  - One part of an EJB runs inside the JVM of a Web application server (such as IBM's WebSphere Application Server).
  - One part runs (typically) inside the JVM of a Web browser.

  An EJB represents either one data row in a database (an *entity bean*), or a connection to a remote database (a *session bean*). Usually, entity beans and session beans are used together, to allow data to be represented and accessed in a standardized way, and in heterogeneous environments that contain both relational and non-relational data. For further information about EJBs, refer to Chapter 20, “Using EJBs to Represent Data,” on page 249.
- *The VSE Java Beans* supplied with the z/VSE e-business Connectors are not visual Java Beans: they conform to the specifications for Java Beans, but are not visual components. Instead, they *represent* z/VSE-based objects such as:
  - File systems (VSE/Librarian, VSE/POWER, VSE/ICCF, VSE/VSAM).
  - System components (such as the operator console).
  - Data objects (such as VSE libraries, POWER queue entries, and VSAM catalogs).

Contents of the VSE Java Beans Class Library

The following table shows the classes of the VSE Java Beans (package *com.ibm.vse.connector*). The Java interfaces are shown in italics.

Table 4. Contents of the VSE Java Beans Class Library

Class	Description
VSECertificateEvent	This class is a event that is specific to an SSL Certificate. It contains the VSEConnectionSpec as source of this event and information about the certificate
VSECertificateListener	This interface must be implemented by any VSECertificateListener. You can register a VSECertificateListener using the method addVSECertificateListener of the VSEConnectionSpec bean. This Listener is used to notify about certificates of a SSL connection.
VSEConnectionManager	Maintains connections to remote VSE systems.
VSEConnectionSpec	This class represents the specification of a connection from your workstation to a z/VSE host. It implements interface ConnectionSpec, which is part of the IBM Common Connector Framework (CCF). CCF maintains a pool of connections that can be reused by different Java programs. This is important especially when writing WebSphere based Java programs, such as servlets, or Enterprise Java Beans. In this case, short living programs, such as servlets, can reuse existing connections to the z/VSE host.
VSEConnectorTrace	This interface must be implemented by any Trace-Class. During startup of the VSE Connector Beans the system tries to load a class called Trace.class in the default package. This class is loaded using Class.forName(). Normally there is no Trace.class available. In this case no trace messages are written. Users can implement its own Trace.class by implementing the interface VSEConnectorTrace. The VSE Connector Beans will call the method writeTrace for each line that is to be written. The Trace class will typically write the trace text to System.out or to a file.
VSEConsole	This class represents the VSE operator console. It allows you to issue console commands, obtain the message output, and get message explanation for a given message number.
VSEConsoleExplanation	This class represents a textual explanation of a given console message. It provides methods to get the message lines and the number of lines.
VSEConsoleMessage	This class represents a console message. It provides methods to get properties, such as message number, color, attributes, and so on.
VSEDli	This class represents the DL/I subsystem on z/VSE. It provides methods to get a list of PSBs.
VSEDliPsb	This class represents a DL/I PSB with its corresponding PSB name. It provides methods to schedule or terminate a PSB, to take a checkpoint or to do a rollback. It also provides a list of PCBs.
VSEDliPcb	This class represents a DL/I PCB which can be used to execute DL/I requests like GN, GNP, GU, GHU, GHN, GHNP, DLET, ISRT, REPL.
VSEIccf	This class represents the VSE/ICCF component of a VSESystem. It provides *read only access* to the VSE/ICCF libraries and members. It provides methods to get a list of VSE/ICCF libraries, search for members, and get properties.
VSEIccfLibrary	This class represents an VSE/ICCF library. It provides methods to get a list of its members, search for members, and so on.
VSEIccfMember	This class represents an ICCF member. It provides methods to copy and download a member, and get properties.
VSELibrarian	This class represents the Librarian of a VSE System. It is required in order to get access to the VSE library system. It provides methods to list libraries and search for members. The list of libraries is specified in the VSE Connector Server configuration file IESLIBDF.

Table 4. Contents of the VSE Java Beans Class Library (continued)

Class	Description
VSELibrary	This class represents a VSE library. It provides methods to access its sub-libraries, search for members, get library properties, and so on. See also VSESubLibrary.
VSELibraryExtent	This class represents extent information of a VSELibrary.
VSELibraryMember	This class represents a VSE library member. It provides methods to copy, delete, download, upload a member instance, get and set properties, and so on.
VSEMessage	This class allows a message from another user to be received.
VSEPlugin	This abstract class is the base class of all user written plugin beans. It allows you to extend the VSE Java Beans class library by your own beans, by implementing the basic functionality of plugins. For a description of how to write a plugin for the VSE Java Beans, refer to "Implementing a Client Plugin" on page 286.
VSEPower	This class represents the VSE/POWER component of a given VSESystem. It provides methods to access the POWER queues, submit jobs, get the related job output, search for queue entries, get and set properties, and so on.
VSEPowerEntry	This class represents one VSE/POWER queue entry. It provides methods to copy, delete, release, download a queue entry, get and set properties, upload a local file into a queue entry, and so on.
VSEPowerQueue	This class represents an instance of a VSE/POWER queue. This can be the reader, list, punch, or transmit queue. It provides methods to get a list of the queue entries, search for queue entries, get queue properties, and so on.
VSEResource	This abstract class is the base class of all VSEResource beans. It implements the basic functionality that must be present for each resource.
VSEResourceEvent	This class represents an event specific to a VSEResource. It contains the source of this event and optionally event related data. It is used, for example, when implementing a VSEResourceListener.
VSEResourceListener	This interface is used to implement callback routines that allow to synchronize sending actions to the host and receiving data objects. A resource listener is called from the low level functions of the VSE Java Beans to notify about received data objects. See online documentation (VSEConnectors.html) for many samples of resource listener implementations.
VSESubLibrary	This class represents a sub-library of a given VSELibrary. It provides methods to create and delete an instance of this class, get a list of members in this sub-library, get and set properties, search for members, and so on. See also VSELibrary.
VSESystem	This class represents a z/VSE host. An instance of this class is needed to connect to the host and get access to the VSE file systems and functionality.
VSEUser	This class represents a user on a given VSE system. It provides methods to check if this user is currently active, get properties and so on.
VSEVsam	This class represents the VSAM component of a given VSESystem. It provides access to the VSAM catalogs and clusters.
VSEVsamCatalog	This class represents a VSAM catalog. It provides methods to get a list of clusters in this catalog, get and set properties.
VSEVsamCluster	This class represents a VSAM file. This can be any cluster, except VRDS. It provides methods to get a list of data maps (see VSEVsamMap), get and set properties, select data records from the cluster, and so on.
VSEVsamField	This class represents a data field of a given VSAM record. A data field represents one specific column of a VSAM record. It is always part of a given VSAM map or view. It consists of a name (the column name), a length, a datatype, and an offset within the record.

Table 4. Contents of the VSE Java Beans Class Library (continued)

Class	Description
VSEVsamFilter	This class represents a filter when getting VSAM data from a VSAM file. It consists of a VSEVsamField, a filter string that may contain wildcards, and boolean operations.
VSEVsamMap	This class represents a data map for a given VSAM record. A map splits the VSAM record into columns, respectively data fields, that have a name, a length, a datatype, and an offset within the record. In addition to data fields, a map can contain data views that are subsets of the map's fields.
VSEVsamRecord	This class represents a data record of a given VSAM file. It provides methods to access the data fields of the record.
VSEVsamView	This class represents a data view on a given VSAM record together with a given data map. A data view always points to a subset of the data fields of a given VSAM map. As a consequence, actions against VSAM maps always influence the views of the related map. For example, deleting a map will also delete all views of this map.

Example of a Javadoc for a VSE Java Bean

Figure 44 shows an example of a Javadoc that was generated from the source code of a VSE Java Bean. The VSE Java Bean belongs to the VSE Java Beans class library.

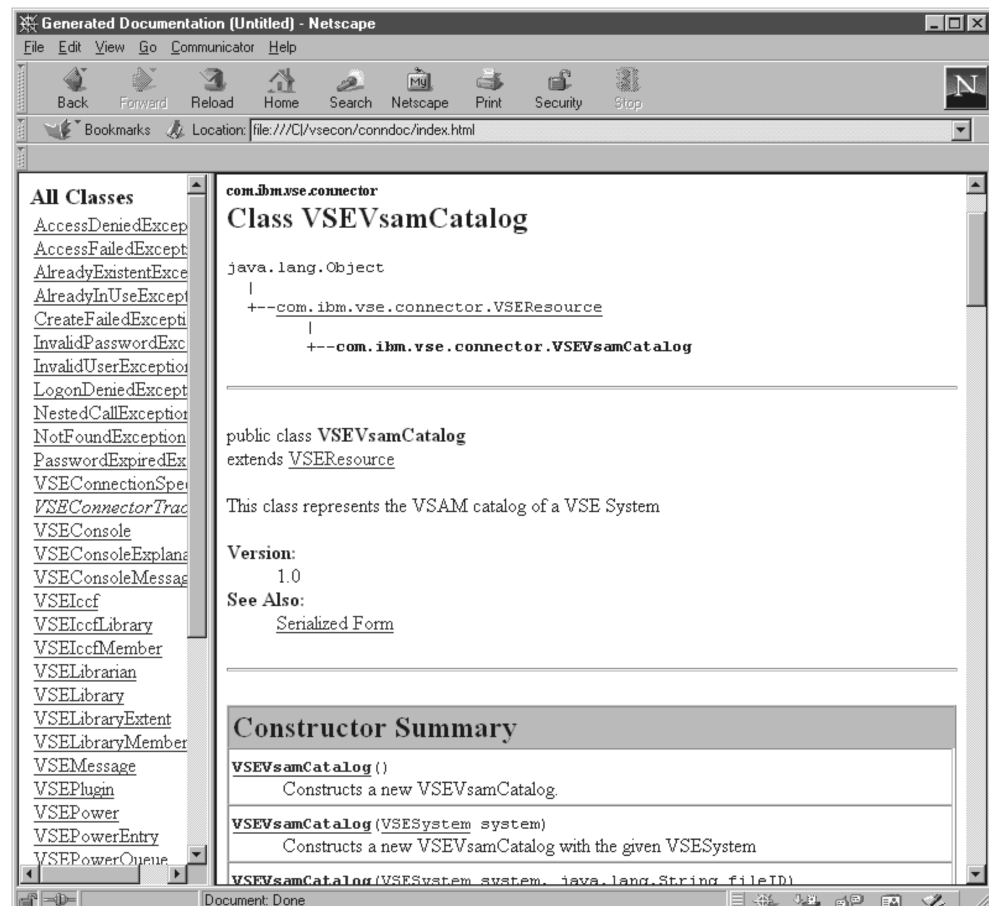


Figure 44. Example of a Javadoc Belonging to the VSE Java Beans Class Library

Note: The online documentation provided with the VSE Connector Client contains much information about the VSE Java Beans class library. See "Using the Online

Documentation Options" on page 26 for details.

Using the Callback Mechanism of VSE Java Beans

You use the *callback mechanism* of VSE Java Beans to:

- Implement a callback function (a *VSEResourceListener*) that is called from the low level functions of the VSE Java Beans, which listen to the host connection, whenever data is received from the host.
- Add the resource listener to the VSE bean from which data shall be received Sending a request for data to the host.
- Get received data in the resource listener implementation, instance by instance.
- Remove the resource listener from the bean.
- Return data from the callback function to the caller.

This topic first describes the callback mechanism in detail, and then describes how to access the various file systems and the operator console. You should also refer to the online documentation for other examples of resource listener implementations.

All VSE Java Beans methods returning lists of objects from the remote VSE host, return data instances through a callback routine. This means, the requesting function (the caller) returns after all data items are received from the host, but each data item can be processed immediately after it is received in the callback routine.

Note: When writing Enterprise Java Beans (EJBs) it is not possible to have callback routines. EJBs are not multi-thread enabled, nor do they support callbacks. Refer to the EJB samples for scenarios where EJBs are appropriate.

The VSE Java Beans class library provides the *VSEResourceListener* callback interface. This Java interface must be implemented by each application that wants to receive lists of VSE resources, such as lists of VSE libraries, sub-libraries, or members, as well as all kind of search results. The interface has only three methods:

listStarted(VSEResourceEvent event)
: Called before the first data block is received from the host. Can be used for initialization purposes. The event does not contain any data.

listAdded(VSEResourceEvent event)
: Called for each instance of a VSE resource, which is contained in the event.

listEnded(VSEResourceEvent event)
: Called after the last data block has been received from the host. Can be used for cleanup purposes. The event does not contain any data.

Here is an example of a *VSEResourceListener* implementation. In addition to the above three required methods, additional methods are implemented which allow received data to be saved, and then returned to the caller.

The example shown below listens for VSAM resources. The example also implements:

- a general listener, that listens to all possible VSE resources.
- specialized listeners, which listen only to some selected resources.

In this part of the example, two vectors are defined to store and accumulate all data objects that are received. As a result, a complete list of the data objects can be returned to the caller.

```
public class VsamListener implements VSEResourceListener
{
   public Vector catVector, fileVector;

   /**
    * constructs a new VsamListener. Two vectors are used to store
    * received resource instances.
    *
    */
   public VsamListener()
   {
      catVector = new Vector();
      fileVector = new Vector();
   }

   /**
    * allows the caller to reset the internal vectors.
    *
    */
   public void clear()
   {
      catVector.removeAllElements();
      fileVector.removeAllElements();
   }

   /**
    * returns the catalog list.
    *
    */
   public Vector getCatalogVector()
   {
      return catVector;
   }

   /**
    * returns the VSAM file list.
    *
    */
   public Vector getFileVector()
   {
      return fileVector;
   }   /**
    * is called for the start of a list of VSEResources before
    * notifying about the list elements.
    * The event does not contain any data.
    *
    * @param event The VSEResourceEvent containing the source
    */
   public void listStarted(VSEResourceEvent event)
   {
      System.out.println("VsamListener: listStarted()");
   }
   /**
    * is called for each element of a list of VSEResources.
    * The VSEResourceEvent contains the data instance (see getData()).
    *
    * @param event The VSEResourceEvent containing the source and the data
    */
   public void listAdded(VSEResourceEvent event)
   {
      VSEResource resource = (VSEResource)(event.getData());
      if (resource instanceof VSEVsamCatalog)
      {
         VSEVsamCatalog cat = (VSEVsamCatalog)resource;
         catVector.addElement(cat);
      }
      else if (resource instanceof VSEVsamCluster)
      {
         VSEVsamCluster file = (VSEVsamCluster)resource;
         fileVector.addElement(file);
      }
```

```
   }
    /**
    * is called for the end of a list of VSEResources after
    * notifying about the list elements. The event does not
    * contain any data.
    *
    * @param event The VSEResourceEvent containing the source
    */
   public void listEnded(VSEResourceEvent event)
   {
      System.out.println("VsamListener: listEnded()");
   }
}
```

A typical flow for getting a list of VSAM catalogs would then look like this:

```
public static void main(String argv[ ]) throws IOException
{
   VSESystem system;       // the VSE host object
   VSEVsam vsam;           // the VSAM object
   VsamListener vl;        // implemented as shown above
   Vector vCatalogs;       // used to store the catalog list

   ...
   vsam = system.getVSEVsam();
   vl = new VsamListener();
   vsam.addVSEResourceListener(vl);
   vsam.getCatalogList();
   vsam.removeVSEResourceListener(vl);
   vCatalogs = vl.getCatalogVector();
   ...
}
```

Figure 45. Program Flow for Using VSE Java Classes to Obtain a List of VSAM Catalogs

Example of Using VSE Java Beans to Connect to the Host

This topic describes the definition of VSE host instances, and the setting up of the connection to the physical z/VSE host.

Each VSE host is represented by an object of class VSESystem. To be able to connect to a physical host, it is required to have a connection specification that holds all properties of the connection. Refer to the online documentation provided with the VSE Connector Client (see “Using the Online Documentation Options” on page 26) for further information about creating and reusing VSE host connections.

By default, a connection is given back to this pool after each host action. There is a method of the *VSESystem* class: *setConnectionMode* (true/false), which allows a connection to be held during the whole lifetime of your application (i.e. to not give it back to the pool after each host access). This should be done when implementing long running applications. The following shows a simple code example of defining a VSESystem and connecting to the host. These two steps are necessary to connect to a VSE host:

Step 1: Create a VSEConnectionSpec

```
/* Create connection specification. The connection spec */
/* holds information about the physical host connection and is */
/* stored permanently in the Common Connector Framework (CCF) */
try {
spec = new VSEConnectionSpec(InetAddress.getByName(ipAddr),
                                             2893,userID,password);
}
catch (UnknownHostException e)
{
   System.out.println("Unknown host : " + e);
   return;
}
spec.setMaxConnections(5);

/* Stay logon with this user for lifetime of this connection */
spec.setLogonMode(true);
```

Figure 46. Connect to Host via VSE Java Beans: Create a VSEConnectionSpec

Step 2: Create a VSESystem

```
/* Create VSE system instance with this connection */
system = new VSESystem(spec);

/* Hold connection for the lifetime of our application */
system.setConnectionMode(true);
system.connect();
```

Figure 47. Connect to Host via VSE Java Beans: Create a VSESystem

Notes:

1. It is not necessary to call the *connect()* method of a *VSESystem* to be able to communicate with the *VSESystem*. Instead, when calling a method that needs a host connection, the connection is opened.
2. For details of how to configure for SSL connections, refer to the chapter “Configuring for Server Authentication” in the *z/VSE Administration*.

Example of Using VSE Java Beans to Connect to the Host via SSL

This example describes *three* possible ways in which you can connect to the z/VSE host via SSL.

Prerequisites for running this example:

- You have submitted the job SKSSLKEY (which you can find in ICCF library 59) to set up the server-side VSE Keyring Library. For further details, refer to the chapter “Preparing Your System to Use SSL” in the *z/VSE Administration*.
- The IBM-provided sample client-side keyring file **Keyring.pfx** is located in the **samples** directory of the VSE Connector Client installation. For further details, see “Performing the VSE Connector Client Installation” on page 26.

Note: Before starting this topic, you should understand the differences between the SSL support offered by IBM's and Sun Microsystems's Java Development Kits. For details, refer to the chapter “Preparing Your System to Use SSL” in the *z/VSE Administration*.

The example begins by specifying the class name and implementing the main method:

```
public class SSLApiExample implements VSECertificateListener
{
   public SSLApiExample() throws IOException, ResourceException
   {
```

The main steps for the rest of the example are shown below.

Step 1: Prompt for IP Address, User ID, Password

```
BufferedReader r = new BufferedReader(
                       new InputStreamReader(System.in));
System.out.println("Please enter your VSE IP address:");
String ipAddr = r.readLine();
System.out.println("Please enter your VSE user ID:");
String userID = r.readLine();
System.out.println("Please enter password:");
String password = r.readLine();
```

Figure 48. Connect to Host via VSE Java Beans and SSL: Prompt for IP Address, User ID, Password

Step 2: Create a Connection Specification for the VSE System

In this step, the instance of *VSESystem* is used to get access to the z/VSE host system. A *VSECertificateListener* object is then used to get access to the digital server certificate, which is sent to this client from the server-side when an SSL connection is established.

```
VSEConnectionSpec spec;
try {
   spec = new VSEConnectionSpec(InetAddress.getByName(ipAddr),
                           2893,userID,password);
}
catch (UnknownHostException e)
{
   System.out.println("Unknown host : " + e);
   return;
}
spec.setLogonMode(true);
spec.setSSL(true);
spec.addVSECertificateListener(this);
```

Figure 49. Connect to Host via SSL and VSE Java Beans: Create a Connection Specification

Specifying the SSL Properties: Alternative 1

In this alternative, the SSL properties are specified *directly*. The properties defined here must match the properties defined for the VSE Connector Server. For details, see job skeleton SKVCSSSL in ICCF library 59 (which is also described in “VSE Library Member SKVCSSSL – Configure for SSL” on page 32).

```
Properties sslProps = new Properties();
sslProps.put("SSLVERSION", "SSL");
sslProps.put("CIPHERSUITES",
     "SSL_RSA_WITH_DES_CBC_SHA," +
     "SSL_RSA_WITH_3DES_EDE_CBC_SHA," +
     "SSL_RSA_WITH_NULL_SHA," +
     "SSL_RSA_WITH_NULL_MD5," +
     "SSL_RSA_EXPORT_WITH_DES40_CBC_SHA"
 );
sslProps.put("KEYRINGFILE", "KeyRing.pfx");
sslProps.put("KEYRINGPWD", "ssltest");
spec.setSSLProperties(sslProps);

/* Create VSE system instance with this connection */
VSESystem system = new VSESystem(spec);
system.connect();
```

Figure 50. Connect to Host via VSE Java Beans and SSL: Specify SSL Properties (Alternative 1)

Specifying the SSL Properties: Alternative 2

In this second alternative, a Java properties file is read to obtain the SSL parameters. The previous *VSEConnectionSpec* instance can be re-used.

```
File file = new File("ssl.prop");
FileInputStream fis = new FileInputStream(file);
DataInputStream in = new DataInputStream(fis);
sslProps = new Properties();
sslProps.load(in);
spec.setSSLProperties(sslProps);
system = new VSESystem(spec);
system.connect();
```

Figure 51. Connect to Host via VSE Java Beans and SSL: Specify SSL Properties (Alternative 2)

Specifying the SSL Properties: Alternative 3

In this third alternative, an existing SSL properties file **ssl.prop** is used directly. This IBM-provided sample file is also located in the **samples** directory.

```
   spec.setSSLPropertiesFile("ssl.prop");
   system = new VSESystem(spec);
   system.connect();

   /* Cleanup and finish */
   spec.removeVSECertificateListener(this);
}
```

Figure 52. Connect to Host via VSE Java Beans and SSL: Specify SSL Properties (Alternative 3)

Main Method of the Class Used in This Example

This is the main method of the class used for this example of how to connect to the z/VSE host via SSL.

```
public static void main(String argv[])
throws IOException, ResourceException
{
   SSLApiExample ex = new SSLApiExample();
}
```

Figure 53. Connect to Host via VSE Java Beans and SSL: Main Method of the Class

Implementation of the ConfirmCertificate Method

The *ConfirmCertificate* method implements the *VSECertificateListener* interface. The event contains various types of data about the server certificate that is received. You can use this information to either:

- Accept the certificate (by simply returning).
- Reject the certificate (by issuing a *CertificateRejectedException*).

So this method is where you can prompt the end-user or display a dialog box to the end-user, and ask the end-user to either accept or reject the certificate.

```
   public void confirmCertificate(VSECertificateEvent event)
   throws CertificateRejectedException
   {
      System.out.println("Received server certificate:");
      System.out.println("Common name   : " + event.getCommonName());
      System.out.println("Organization  : " + event.getOrganization());
      System.out.println("Org. unit     : " + event.getOrganizationUnit());
      System.out.println("Issuer name   : " + event.getIssuerCommonName());
      System.out.println("Issuer Org.   : " + event.getIssuerOrganization());
      System.out.println("Issuer O-unit : " + event.getIssuerOrganizationUnit());
      System.out.println("Public key    : " + event.getPublicKey());
      System.out.println("Serial number : " + event.getSerialNumber());
      System.out.println("Valid from    : " + event.getValidFrom());
      System.out.println("Valid to      : " + event.getValidTo());
      System.out.println("Verified?     : " + event.isVerified());
      //throw new CertificateRejectedException("certificate rejected.");
   }
}
```

Figure 54. Connect to Host via VSE Java Beans and SSL: Implementation of ConfirmCertificate Method

Example of Using VSE Java Beans to Submit Jobs to the Host

This example includes code parts that are taken from the *JobApiExample*, described in the online documentation. The example shows how to submit jobs, and track the status of submitted jobs. Two methods of doing so are described:

- The easiest way is to create a file on the local hard disk that contains the complete VSE/POWER job. This file is sent to the host, the job output is received into another local file.
- A second way, that does not need access to the local file system, is to create the JCL in memory and also receive the job output in memory line by line.

Note: Refer to the online documentation provided with the VSE Connector Client (see "Using the Online Documentation Options" on page 26) for details of how to create and re-use z/VSE host connections.

The example begins by specifying the class name and implementing the main method:

```
public class JobApiExample
{
   public static void main(String argv[]) throws IOException, ResourceException
   {
```

The main steps for the rest of the example are shown below.

Step 1: Prompt for IP Address, User ID, Password

See *page "Step 1: Prompt for IP Address, User ID, Password" on page 150* (the code is the same as for connecting to the z/VSE host via VSE Java Beans).

Step 2: Create a Connection Specification for the VSE System

In this step, the instance of *VSESystem* is used to get access to the VSE file systems (which in this example is VSE/POWER).

```
VSEConnectionSpec spec;
try {
   spec = new VSEConnectionSpec(InetAddress.getByName(ipAddr),
                                        2893, userID, password);
}
catch (UnknownHostException e)
{
   System.out.println("Unknown host : " + e);
   return;
}
spec.setLogonMode(true);
VSESystem system = new VSESystem(spec);
system.setConnectionMode(true);
VSEPower power = system.getVSEPower();
```

Figure 55. Submit Jobs via VSE Java Beans: Create a Connection Specification

Step 3: Submit a Job File

In this step, a job file is submitted that is stored on the local disk in file **test.job**. The *execute()* method returns as soon as the job output has been transferred from the POWER list queue to the outFile **out.txt** in the current directory.

```
File jobFile = new File("test.job");
File outFile = new File("out.txt");
power.executeJob(jobFile, outFile);
```

Figure 56. Submit Jobs via VSE Java Beans: Submit a Job File

Step 4: Create the Job File and Send It to the Host

In this step, the job file is created in memory and is then sent to the z/VSE host. This has the advantage that access to the local file system is not required. However, it is necessary to implement:

- A *JobInputStream*, which contains the JCL.
- A *JobOutputStream*, which receives the job output.

For further details, refer to the Java sources *JobInputStream.java* and *JobOutputStream.java*.

```
      JobInputStream job = new JobInputStream();
      JobOutputStream dest = new JobOutputStream();
      power.executeJob(job, dest);
      Vector vLines = dest.getAllLines();
      System.out.println(
         "Number of output lines: " + new Integer(vLines.size()).toString());
      for (int i=0;i<vLines.size();i++)
      {
         System.out.println((String)(vLines.elementAt(i)));
      }
   }
```

Figure 57. Submit Jobs via VSE Java Beans: Create the Job File and Send to the Host

Here are the main parts of the *JobInputStream* class:

1. Specify the class name.

```
public class JobInputStream implements DataInput
{
   int linesRead;
   int maxLines;
```

2. The complete VSE/POWER job is contained in a local String array. It is therefore possible to assign job parameters, such as class or disposition, dynamically by:
   a. Passing these parameters in the constructor.
   b. Modifying the String array.

```
protected String[] jclArray = {
  "* $$ JOB JNM=TEST2,CLASS=0,DISP=D",
  "* $$ LST CLASS=A,DISP=L,PRI=3,LST=SYSLST",
  "// JOB TEST2",
  "// EXEC LIBR",
  " ACC S=IJSYSRS.SYSLIB",
  " LD INW*.PHASE",
  "/*",
  "/&",
  "* $$ EOJ"
};
```

3. The constructor for the *JobInputStream* class is shown below:

```
public JobInputStream()
{
   super();
   linesRead = -1;
   maxLines = jclArray.length;
}
...
```

4. The callback routine is shown below. It is called by the job execution routine *executeJob()* of the *VSEPower* instance, to obtain the next JCL line and submit it to the connected VSE system. Therefore, this routine is called once for each JCL line.

```
   public String readLine()
   {
      if (linesRead < maxLines-1)
      {
         linesRead++;
         return jclArray[linesRead];
      }
      else
         return null;
   }
}
```

Example of Using VSE Java Beans to Access the Operator Console

There are two ways of how to get console messages when sending a console command:

1. Using the *execute()* method of class *VSEConsole*, returns a complete list of messages.
2. Using *open()*, *setCommand()*, *getMessage()*, and *close()*, allows you to work with the first messages while others are still being retrieved.

This example includes code parts that are taken from the *ConsoleApiExample*, which is described in detail in the online documentation.

Note: Refer to the online documentation provided with the VSE Connector Client (see "Using the Online Documentation Options" on page 26) for details of how to create and re-use z/VSE host connections.

The example begins by specifying the class name and implementing the main method:

```
public class ConsoleApiExample
{
   public static void main(String argv[]) throws IOException, ResourceException
   {
```

The main steps for the rest of the example are shown below.

Step 1: Prompt for IP Address, User ID, Password

See *page "Step 1: Prompt for IP Address, User ID, Password" on page 150* (the code is the same as for connecting to the z/VSE host via VSE Java Beans).

Step 2: Create a Connection Specification for the VSE System

In this step, the instance of *VSESystem* is used to get access to the operator console.

```
VSEConnectionSpec spec;
try {
   spec = new VSEConnectionSpec(InetAddress.getByName(ipAddr),
                            2893, userID, password);
}
catch (UnknownHostException e)
{
   System.out.println("Unknown host : " + e);
   return;
}
spec.setLogonMode(true);
VSESystem system = new VSESystem(spec);
system.setConnectionMode(true);
```

Figure 58. Access Console via VSE Java Beans: Create a Connection Specification

Step 3: Create a Console Instance and Send a Command

In this step, the command output is returned in vector *vConsOutput*. It is important here to specify an end-string as the third parameter in *cons.execute()*. Otherwise, the function would be unable to return an end-condition immediately after the last message.

```
VSEConsole cons = new VSEConsole(system);
Vector vConsOutput = cons.execute("map", null, "AR 0015 1I40I  READY", 30);
for (int i=0;i<vConsOutput.size();i++)
   System.out.println(
      ((VSEConsoleMessage)(vConsOutput.elementAt(i))).getMessage());
```

Figure 59. Access Console via VSE Java Beans: Create Console Instance, Send a Command

Step 4: Obtain Messages One Line at-a-Time

In this step, the command output is returned as a series of messages that are sent individually. To do so, functions *open()* and *setCommand()* are used.

```
      cons.open();
      cons.setCommand("map");
      VSEConsoleMessage message = cons.getMessage();
      boolean finished = false;
      while (!finished)
      {
         if (message != null)
         {
            System.out.println(message.getMessage());
            if (message.getMessage().indexOf("AR 0015 1I40I  READY") >= 0)
               finished = true;
         }
         message = cons.getMessage();
      }
      cons.close();
   }
}
```

Figure 60. Access Console via VSE Java Beans: Obtain Messages One Line at-a-Time

Example of Using VSE Java Beans to Access VSAM Data

This example includes parts taken from the *VsamDisplayExample*, which is described in detail in the online documentation. The example displays the data of a VSAM file using a map, and adds a new record to a file. It assumes that there is a VSAM cluster FLIGHT.ORDERING.FLIGHTS containing a map FLIGHTS_MAP that describes the columns (data fields) of the record.

This example can only be run providing:
1. The VSAM files FLIGHT.ORDERING.FLIGHTS (KSDS) and FLIGHTS.ORDERING.ORDERS (RRDS) have been defined.
2. The above VSAM files contain sample data.

For details on how to define these VSAM files and load sample data, see "Creating the VSAM Clusters for the Sample" on page 231.

Note: Refer to the online documentation provided with the VSE Connector Client (see "Using the Online Documentation Options" on page 26) for details of how to create and re-use z/VSE host connections.

The example begins by specifying the class name and implementing the main method:

```
public class VsamDisplayExample
{
   public static void main(String argv[]) throws IOException, ResourceException
   {
```

The main steps for the rest of the example are shown below.

Step 1: Define the Local Variables

```
String catName  = "VSESP.USER.CATALOG";
String fileName = "FLIGHT.ORDERING.FLIGHTS";
String mapName  = "FLIGHTS_MAP";
```

Figure 61. VSAM Data via VSE Java Beans: Define Local Variables

Step 2: Prompt for IP Address of VSE System, User ID, Password

See *page "Step 1: Prompt for IP Address, User ID, Password" on page 150* (the code is the same as for connecting to the z/VSE host via VSE Java Beans).

Step 3: Create Connection Specification for the VSE System

In this step, the instance of *VSESystem* is used to get access to the VSE file systems (which in this example is VSAM).

```
VSEConnectionSpec spec;
try {
   spec = new VSEConnectionSpec(InetAddress.getByName(ipAddr),
                                 2893,userID,password);
}
catch (UnknownHostException e)
{
   System.out.println("Unknown host : " + e);
   return;
}
spec.setLogonMode(true);
VSESystem system = new VSESystem(spec);
system.setConnectionMode(true);
VSEVsam vsam = system.getVSEVsam();
```

Figure 62. VSAM Data via VSE Java Beans: Create Connection Specification

Step 4: Create a VSEResourceListener

In this step, a *VSEResourceListener* is created that is notified each time an object is retrieved from the z/VSE host. For further details, refer to the Java source file *RecordListener.java*.

```
RecordListener rl = new RecordListener();
vsam.addVSEResourceListener(rl);
```

Figure 63. VSAM Data via VSE Java Beans: Create a VSEResourceListener

Step 5: Get VSAM Records from the z/VSE Host

In this step, VSAM records are retrieved from the z/VSE host using the map provided. The *selectRecords()* method performs the data access using this map. All records can then be accessed using the vector *vRecords*.

```
VSEVsamMap map = new VSEVsamMap(system, catName, fileName, mapName);
VSEVsamCluster cluster = new VSEVsamCluster(system, catName, fileName);
cluster.addVSEResourceListener(rl);
cluster.selectRecords(map);
cluster.removeVSEResourceListener(rl);
Vector vRecords = rl.getRecords();
```

Figure 64. VSAM Data via VSE Java Beans: Get VSAM Records from Host

Step 6: Display VSAM Records

In this step, *two* loops are used:

- An outer loop used for obtaining the list of VSAM records (contained in vector *vRecords*).
- An inner loop used for obtaining the data fields (columns) of each record.

```
VSEVsamRecord record;
int numMapFields = map.getNoOfFields();
for (int k=0;k<vRecords.size();k++)
{
   System.out.println("Record " + k + ":");
   record = (VSEVsamRecord)(vRecords.elementAt(k));
   for (int i=0;i<numMapFields;i++)
   {
      try {
         System.out.println(map.getFieldName(i) + " : " + record.getField(i));
      }
      catch (Exception e)
      {
         ...
      }
   }
}
```

Figure 65. VSAM Data via VSE Java Beans: Display VSAM Records

Step 7: Insert a VSAM Record in the VSAM Cluster

In this step, a VSAM record (a new flight) is inserted in the VSAM cluster. If the flight number already exists in the VSAM KSDS cluster called FLIGHT.ORDERING.FLIGHTS, an exception occurs.

```
boolean done = false;
VSEVsamRecord newRec;
while (!done)
{
   System.out.println("Please enter a new flight number:");
   System.out.println("(Enter a negative value to quit)");
   String flightNum = r.readLine();
   if ((new Integer(flightNum).intValue()) < 0)
      return;

   /* Check if new flight number already exists ... */
   try {
      newRec = new VSEVsamRecord(system, catName, fileName, mapName);
      newRec.setKeyField(0, new Integer(flightNum));
      newRec.add();
      done = true;
   }
   catch (Exception e)
   {
      System.out.println("Exception when adding new record.");
      if (e instanceof AlreadyExistentException)
         System.out.println("This flight number already exists.");
      else
         ...
   }
}
```

Figure 66. VSAM Data via VSE Java Beans: Insert a VSAM Record

Step 8: Prompt the User to Enter Column Values

In this step, the user is prompted to enter column values for the VSAM record that was inserted during Step 7.

```
      for (int i=1;i<numMapFields;i++)
      {
         System.out.println("Please enter value for field " + map.getFieldName(i) +
                            " (Length = " + map.getFieldLength(i) + ")");
         String newField = r.readLine();
         if (map.getFieldType(i) == VSEVsamField.TYPE_STRING)
            newRec.setField(i, newField);
         else
            newRec.setField(i, new Integer(newField));
      }

      /* Make changes permanent... */
      newRec.commit();
   }
}
```

Figure 67. VSAM Data via VSE Java Beans: Prompt the User for Values

Example of Using VSE Java Beans to Access DL/I Data

This example includes parts taken from the *DliApiExample.Java*, which is described in detail in the online documentation. The example displays the data of a DL/I database. It assumes that the sample DL/I database has been defined and loaded, as described in Chapter 6, “Configuring DL/I for Access Via VSE Java Beans,” on page 37.

Previous to VSE/ESA 2.7, DL/I data could only be accessed via a DB2 Stored Procedure. From VSE/ESA 2.7 onwards, DL/I data can be accessed without using DB2. Instead of using DB2, the DL/I data is accessed using the VSE Connector Server, which then uses the AIBTDLI interface to access DL/I.

Note: Refer to the online documentation provided with the VSE Connector Client (see "Using the Online Documentation Options" on page 26) for details of how to create and re-use z/VSE host connections.

Step 1: Create a VSE System Instance and Get Access to DL/I

```
public class DliApiExample
{
  protected static VSEConnectionSpec spec;
  protected static VSESystem system;
  protected static VSEDliPsb psb = null;
  protected static VSEDliPcb pcb = null;

  public static void main(String argv[])
  {
    ...
    byte[] ioarea;
    String[] ssas;

    try
    {
      ...

      /* Create VSE system ... */
      system = new VSESystem(spec);

      /* Get the DLI subsystem */
      VSEDli dli = new VSEDli(system);
```

Figure 68. DL/I Data via VSE Java Beans: Get Access to DL/I

Step 2: Schedule the PSB

In this step, the PSB (Program Specification Block) is scheduled.

```
psb = dli.getDliPsb("STBICLG");
psb.schedule();
System.out.println("  Num PCBs: " + psb.getNumberOfPCBs());
System.out.println("  IO len:   " + psb.getMaxLengthOfIOArea());
```

Figure 69. DL/I Data via VSE Java Beans: Schedule the PSB

Step 3: Get a PCB

In this step, a PCB (Program Communication Block) is obtained, which in this case is the first PCB in the PSB.

```
pcb  = psb.getVSEDliPcb(0);
System.out.println("  DBDName = " + pcb.getDBDName());
```

Figure 70. DL/I Data via VSE Java Beans: Get a PCB

Step 4: List DL/I Segments

In this step, all DL/I segments are listed. The segments have the data layout shown below (as defined in a COBOL copybook).

```
/* 01  STPIITM         REDEFINES IOAREA.
      02  ITNUMB    PIC X(6).
      02  ITDESC    PIC X(25).
      02  IQOH      PIC X(6).
      02  IQOR      PIC X(6).
      02  FILLER    PIC X(6).
      02  IUNIT     PIC 9(6).
      02  FILLER    PIC X(105).  */

ssas = new String[1];
ssas[0] = "STPIITM  ";
do
{
  /* Get the next segment */
  ioarea = pcb.call("GN", null, ssas);
  System.out.println("  Status = " + pcb.getStatus());
  if (!pcb.getStatus().equals("  "))
    break;
}
while(true);
```

Figure 71. DL/I Data via VSE Java Beans: List DL/I Segments

Step 5: Insert or Update a DL/I Segment

In this step, a segment is inserted or updated. A GHU call is first issued, to check if the segment exists. If the segment does exist, it is updated. If the segment does not exist, a new segment is inserted.

```
String item = "000700";  // item number to insert/update
ssas = new String[1];
ssas[0] = "STPIITM (STQIINO = " + item + ")";

/* Get the segment */
ioarea = pcb.call("GHU", null, ssas);
if (pcb.getStatus().equals("GB") || pcb.getStatus().equals("GE"))
{
   // allocate a new ioarea
   ioarea = new byte[psb.getMaxLengthOfIOArea()];
   for (int i=0;i<ioarea.length;i++)
     ioarea[i] = 0x00;

   // set the new values into the IOArea
   VSEDliPcb.setStringToBuffer(ioarea,item,0,6);
   VSEDliPcb.setStringToBuffer(ioarea,"INSERTED ITEM",6,25);
   VSEDliPcb.setStringToBuffer(ioarea,"000001",31,6);
   VSEDliPcb.setStringToBuffer(ioarea,"000002",37,6);
   VSEDliPcb.setStringToBuffer(ioarea,"000000",43,6);
   VSEDliPcb.setStringToBuffer(ioarea,"000003",49,6)

   // do the insert
   ssas = new String[1];
   ssas[0] = "STPIITM  ";
   pcb.call("ISRT", ioarea, ssas);
}
else
{
   // segment already existing, do an update
   // set the new values into the IOArea
   VSEDliPcb.setStringToBuffer(ioarea,item,0,6);
   VSEDliPcb.setStringToBuffer(ioarea,"UPDATED ITEM",6,25);
   VSEDliPcb.setStringToBuffer(ioarea,"000004",31,6);
   VSEDliPcb.setStringToBuffer(ioarea,"000005",37,6);
   VSEDliPcb.setStringToBuffer(ioarea,"000000",43,6);
   VSEDliPcb.setStringToBuffer(ioarea,"000006",49,6);

   // do the update
   ssas = new String[1];
   ssas[0] = "STPIITM  ";
   pcb.call("REPL", ioarea, ssas);
}
```

Figure 72. DL/I Data via VSE Java Beans: Insert/Update a DL/I Segment

Step 6: Delete a DL/I Segment

```
item = "000700";  // item number to delete

// first verify if the segment is existing
ssas = new String[1];
ssas[0] = "STPIITM (STQIINO = " + item + ")";

/* Get the segment */
ioarea = pcb.call("GHU", null, ssas);
System.out.println("  Status = " + pcb.getStatus());

if (pcb.getStatus().equals("GB") || pcb.getStatus().equals("GE"))
{
  // segment not found
  System.out.println("  Segment not found");
}
else
{
  // segment exists, delete it
  ssas = new String[1];
  ssas[0] = "STPIITM  ";
  pcb.call("DLET", null, ssas);
}
```

Figure 73. DL/I Data via VSE Java Beans: Delete a DL/I Segment

Step 7: Terminate the PSB

```
  psb.terminate();
}
...
```

Figure 74. DL/I Data via VSE Java Beans: Terminate the PSB

Example of Using VSE Java Beans to Access VSE/POWER Data

This example includes code parts that are taken from the *PowerApiExample*, which is shown in detail in the *online documentation* (see page "Using the Online Documentation Options" on page 26). Basically we deal with instances of *VSEPower*, *VSEPowerQueue*, and *VSEPowerEntry*. The example downloads a job from the reader queue and searches for list queue entries containing a given text string.

Note: Refer to the online documentation provided with the VSE Connector Client (see "Using the Online Documentation Options" on page 26) for details of how to create and re-use z/VSE host connections.

The example begins by specifying the class name and implementing the main method:

```
public class PowerApiExample
{
   public static void main(String argv[]) throws IOException, ResourceException
   {
```

The main steps for the rest of the example are shown below.

Step 1: Prompt for IP address, User ID, and Password

See *page "Step 1: Prompt for IP Address, User ID, Password" on page 150* (the code is the same as for connecting to the z/VSE host via VSE Java Beans).

Step 2: Create a Connection Specification for the VSE System

In this step, a connection specification for the z/VSE host is created. The instance of *VSESystem* is used to get access to the VSE file systems (in this example, to VSE/POWER and the VSE/POWER reader queue).

```
VSEConnectionSpec spec;
try {
   spec = new VSEConnectionSpec(InetAddress.getByName(ipAddr),
                             2893, userID, password);
}
catch (UnknownHostException e)
{
   System.out.println("Unknown host : " + e);
   return;
}
spec.setLogonMode(true);
VSESystem system = new VSESystem(spec);
system.setConnectionMode(true);
VSEPower power = system.getVSEPower();
VSEPowerQueue readerQueue = power.getReaderQueue();
```

Figure 75. VSE/POWER Data via VSE Java Beans: Create a Connection Specification

Step 3: Create a VSEResourceListener

In this step, a *VSEResourceListener* is created that is notified each time an object is retrieved from the z/VSE host. See Java source file *PowerListener.java* for details. Method *getEntryList()* gets all entries in the reader queue with class = 0. Then a search is made for the PAUSEBG job, which is downloaded to a local file.

```
PowerListener pl = new PowerListener();
readerQueue.addVSEResourceListener(pl);
readerQueue.getEntryList("*", "*", '0');
readerQueue.removeVSEResourceListener(pl);

/* Now get the entry list from the listener */
Vector vEntries = pl.getEntryVector();
VSEPowerEntry entry;
for (int i=0;i<vEntries.size();i++)
{
   entry = (VSEPowerEntry)(vEntries.elementAt(i));

   /* Look for the PAUSEBG job and download it */
   if (entry.getName().equals("PAUSEBG"))
   {
      File tempFile = new File("pausebg.job");
      entry.get(tempFile);
   }
}
```

Figure 76. VSE/POWER Data via VSE Java Beans: Create a VSEResourceListener

Step 4: Scan Compile Outputs for Errors

In this step, the compile outputs in the list queue are scanned for errors. It is assumed here that a compile job called `compjob` was submitted in F4. The *search()* function then searches for all list queue entries that belong to `compjob` in class = 4, and which contains the string `==ERROR`.

```
      VSEPowerQueue listQueue = power.getListQueue();
      listQueue.addVSEResourceListener(pl);
      pl.clearVector();
      listQueue.search("compjob", userID, '4', "==ERROR", true);
      listQueue.removeVSEResourceListener(pl);

      /* Get the results from the listener */
      vEntries = pl.getEntryVector();
      if (vEntries.size() == 0)
         System.out.println("No list queue entries contain the string
                            \"==ERROR\".");

      for (int i=0;i<vEntries.size();i++)
      {
         entry = (VSEPowerEntry)(vEntries.elementAt(i));
         System.out.println("String found in : " + entry.getName() + "." +
                            entry.getNumber() + "[" + entry.getSuffix() + "]");
      }
   }
}
```

Figure 77. VSE/POWER Data via VSE Java Beans: Scan for Compile Errors

Example of Using VSE Java Beans to Access Librarian Data

This example includes code parts taken from the *LibrApiExample*, which is described in detail in the online documentation. The example gets a list of VSE libraries, then gets a list of sub-libraries within PRD2, then gets a list of members within PRD2.CONFIG. Finally it downloads the first member in PRD2.CONFIG to the local hard disk.

Note: Refer to the online documentation provided with the VSE Connector Client (see “Using the Online Documentation Options” on page 26) for details of how to create and re-use z/VSE host connections.

The example begins by specifying the class name and implementing the main method:

```
public class PowerApiExample
{
   public static void main(String argv[]) throws IOException, ResourceException
   {
```

The main steps for the rest of the example are shown below.

Step 1: Prompt for IP address, User ID, and Password

See *page “Step 1: Prompt for IP Address, User ID, Password” on page 150* (the code is the same as for connecting to the z/VSE host via VSE Java Beans).

Step 2: Create a Connection Specification for the VSE System

In this step, a connection specification for the z/VSE host is created. The instance of *VSESystem* is used to get access to the VSE file systems (in this example, to Librarian).

```
VSEConnectionSpec spec;
try {
spec = new VSEConnectionSpec(InetAddress.getByName(ipAddr),
                          2893,userID,password);
}
catch (UnknownHostException e)
{
   System.out.println("Unknown host : " + e);
   return;
}
spec.setLogonMode(true);
VSESystem system = new VSESystem(spec);
system.setConnectionMode(true);
VSELibrarian libr = system.getVSELibrarian();
```

Figure 78. Librarian Data via VSE Java Beans: Create a Connection Specification

Step 3: Create a VSEResourceListener

In this step, a *VSEResourceListener* is created that is notified each time an object is retrieved from the z/VSE host. Next, a list of VSE libraries is obtained from this z/VSE host. This *getLibraryList()* method will return control after all resources have been received (that is, when the *listEnded()* method of the listener has been called). You are recommended to remove the resource listener from the VSE resource object after the *getLibraryList()* request has returned control. Otherwise, if you later use this listener for another object (for example for a sub-library), the previous object will also be notified.

```
LibrListener ll = new LibrListener();
libr.addVSEResourceListener(ll);
libr.getLibraryList();
libr.removeVSEResourceListener(ll);
```

Figure 79. Librarian Data via VSE Java Beans: Create a VSEResourceListener

Step 4: Obtain a List of Libraries From the VSEResourceListener

In this step, a list of libraries is obtained from the *VSEResourceListener* object. The loop is left after the PRD2 library instance has been obtained. The variable *myLib* then points to an object which represents the PRD2 library.

```
Vector vLibs = ll.getLibVector();
int numLibs = vLibs.size();
VSELibrary myLib;
for (int i=0;i<vLibs.size();i++)
{
   myLib = (VSELibrary)(vLibs.elementAt(i));
   if (myLib.getName().equals("PRD2"))
      break;
}
```

Figure 80. Librarian Data via VSE Java Beans: Obtain a List of Libraries

Step 5: Obtain and Count a List of Sub-Libraries

In this step, a list of sub-libraries is obtained from the *VSEResourceListener* object, and counted.

```
myLib.addVSEResourceListener(ll);
myLib.getSubLibraryList();
myLib.removeVSEResourceListener(ll);
int numSublibs = myLib.getNumberOfSublibs();
```

Figure 81. Librarian Data via VSE Java Beans: Obtain/Count a List of Sub-Libraries

Step 6: Obtain the Instance of the PRD2.CONFIG Sub-Library

In this step, the instance of the PRD2.CONFIG sub-library is obtained from the vector. The loop is left when the sub-library name is CONFIG. Variable *mySublib* then points to an object which represents the PRD2.CONFIG sub-library.

```
VSESubLibrary mySublib;
Vector vSublibs = ll.getSublibVector();
for (int i=0;i<vSublibs.size();i++)
{
   mySublib = (VSESubLibrary)(vSublibs.elementAt(i));
   if (mySublib.getName().equals("CONFIG"))
      break;
}
```

Figure 82. Librarian Data via VSE Java Beans: Get the Instance of the Sub-library

Step 7: Obtain a List of Members in the PRD2.CONFIG Sub-Library

In this step, a list is obtained of the members in the PRD2.CONFIG sub-library. Again, this is done using an instance of *VSEResourceListener*.

```
mySublib.addVSEResourceListener(ll);
mySublib.getMemberList();
mySublib.removeVSEResourceListener(ll);
Vector vMembers = ll.getMemberVector();
int numMembers = mySublib.getNumberOfMembers();
```

Figure 83. Librarian Data via VSE Java Beans: Obtain a List of Members in PRD2.CONFIG

Step 8: Obtain Properties of the First Member

In this step, some of the properties of the first member are obtained. For each property there is a *get...()* method. For properties that can be changed, there is also a *set...()* method that allows the property to be changed.

```
VSELibraryMember myMember
if (vMembers.size() < 0)
{
   myMember = (VSELibraryMember)(vMembers.elementAt(0));
   int num = myMember.getNumberOfRecords();
   int len = myMember.getLogicalRecordLength();
   Calendar creation = myMember.getCreation();
   Calendar update = myMember.getLastUpdate();
   ...
}
```

Figure 84. Librarian Data via VSE Java Beans: Obtain a List of Members in PRD2.CONFIG

Step 9: Download the Member to a Local Disk

In this last step, the member is downloaded to the local “hard” disk. The *download()* method returns control after the download is complete. There are several ways to download a member:

- You can download into a local file.
- You can download to a *DataOutputStream* that does not write any data to disk. Instead you can obtain the file contents line-by-line into storage.

```
      File localFile = new File(myMember.getName() + "." + myMember.getType());
      myMember.download(localFile);
   }
}
```

Figure 85. Librarian Data via VSE Java Beans: Download the Member to Disk

Example of Using VSE Java Beans to Access VSE/ICCF Data

The example provided here includes code parts that are taken from the *IccfApiExample*, which is described in detail in the online documentation. The example:

1. Creates a host and gets a list of VSE/ICCF libraries.
2. Downloads member C$QCNBAT (a compile skeleton) out of VSE/ICCF library 2.
3. Displays some of the member's properties.

Notes:

1. Access to VSE/ICCF is read only.
2. Refer to the online documentation provided with the VSE Connector Client (see “Using the Online Documentation Options” on page 26) for details of how to create and re-use z/VSE host connections.

The example begins by specifying the class name and implementing the main method:

```
public class IccfApiExample
{
   public static void main(String argv[]) throws IOException, ResourceException
   {
```

The main steps for the rest of the example are shown below.

Step 1: Prompt for IP address, User ID, and Password

See *page “Step 1: Prompt for IP Address, User ID, Password” on page 150* (the code is the same as for connecting to the z/VSE host via VSE Java Beans).

Step 2: Create a Connection Specification for the VSE System

In this step, a connection specification for the z/VSE host is created. The instance of *VSESystem* is used to get access to the VSE file systems (in this example, to VSE/ICCF).

Access to VSE/ICCF data is read-only because the VSE Connector Server uses DTSUTIL jobs to get information from the DTSFILE. Furthermore, it was decided not to disconnect the DTSFILE for write-access to avoid disruption to other users of the system.

```
VSEConnectionSpec spec;
try {
spec = new VSEConnectionSpec(InetAddress.getByName(ipAddr),
                          2893,userID,password);
}
catch (UnknownHostException e)
{
   System.out.println("Unknown host : " + e);
   return;
}
spec.setLogonMode(true);
VSESystem system = new VSESystem(spec);
system.setConnectionMode(true);
VSEIccf iccf = system.getVSEIccf();
```

Figure 86. VSE/ICCF Data via VSE Java Beans: Create a Connection Specification

Step 3: Create a VSEResourceListener

In this step, a *VSEResourceListener* is created that is notified each time an object is retrieved from the z/VSE host. Next, a list of VSE/ICCF libraries is obtained from this z/VSE host. This *getLibraryList()* method will return control after all resources have been received (that is, when the *listEnded()* method of the listener has been called). You are recommended to remove the resource listener from the VSE resource object after the *getLibraryList()* request has returned control.

```
IccfListener il = new IccfListener();
iccf.addVSEResourceListener(il);
iccf.getLibraryList();
iccf.removeVSEResourceListener(il);
```

Figure 87. VSE/ICCF Data via VSE Java Beans: Create a VSEResourceListener

Step 4: Obtain a List of ICCF Libraries From the VSEResourceListener

In this step, a list of VSE/ICCF libraries is obtained from the *VSEResourceListener* object. A compile skeleton is then downloaded from VSE/ICCF library 2.

```
Vector vLibs = il.getLibVector();
VSEIccfLibrary myLib;
VSEIccfMember myMember;
for (int i=0;i<vLibs.size();i++)
{
   myLib = (VSEIccfLibrary)(vLibs.elementAt(i));
   if (myLib.getLibrary() == 2)
   {
      /* Get memberlist of ICCF lib 2 */
      myLib.addVSEResourceListener(il);
      myLib.getMemberList();
      myLib.removeVSEResourceListener(il);
      Vector vMembers = il.getMemberVector();
      for (int j=0;j<vMembers.size();j++)
      {
         myMember = (VSEIccfMember)(vMembers.elementAt(j));
         if (myMember.getName().equals("C$QCNBAT"))
         {
            myMember.download("C$QCNBAT.SKL");
         }
      }
   }
}
```

Figure 88. VSE/ICCF Data via VSE Java Beans: Obtain a List of ICCF Libraries

Step 5: Download a Specific VSE/ICCF Member

In this step, a fast way to download a specific VSE/ICCF member is described. The member is searched for, and then directly downloaded.

```
iccf.addVSEResourceListener(il);
il.clear();
iccf.search(2, "C$QCNBAT", "*");
iccf.removeVSEResourceListener(il);
vMembers = il.getMemberVector();
if (vMembers.size() == 1)
{
   myMember = (VSEIccfMember)(vMembers.elementAt(0));
   myMember.download("C$QCNBAT.SKL2");
}
```

Figure 89. VSE/ICCF Data via VSE Java Beans: Download a Specific Member

Step 6: Download a Specific VSE/ICCF Member (Very Fast Method)

In this step, a *very* fast way to download a specific VSE/ICCF member is described. A member object is simply created, and then an attempt is made to download it. If the VSE/ICCF member does not exist on the z/VSE host, an exception is received.

```
      try {
         myMember = new VSEIccfMember(system, 2, "C$QCNBAT");
         myMember.download("C$QCNBAT.SKL3");
      }
      catch (Exception e)
      {
         ...
      }
   }
}
```

Figure 90. VSE/ICCF Data via VSE Java Beans: Download a Specific Member (Very Fast)

Using the VSE Navigator Application

The *VSE Navigator* is a Java application that uses virtually all the VSE Java Beans shown in Table 4 on page 143. It implements a graphical user interface (GUI) that has a similar appearance to many of the currently-available file managers.

The *client-part* of the VSE Navigator, which communicates with the *VSE Connector Server*, provides you with a variety of functions. You can, for example:

- Access VSE file systems (POWER, Librarian, ICCF, VSAM).
- Create and submit jobs, including generating jobs based upon the skeletons stored in ICCF library 2.
- Work with the VSE operator console.
- Compare files, and perform a full-text search in VSE-based file systems.
- Interactively insert and edit VSAM records.
- Display:
  - the VSE hardware configuration, including the property dialogs for attached devices
  - the VSE system activity (CPU usage, and so on)
  - the current VSE service level

– the system labels
– the system tasks
– the used and free VSAM space
– VSAM data, via maps and views

Here are two examples of the GUI provided by the VSE Navigator:

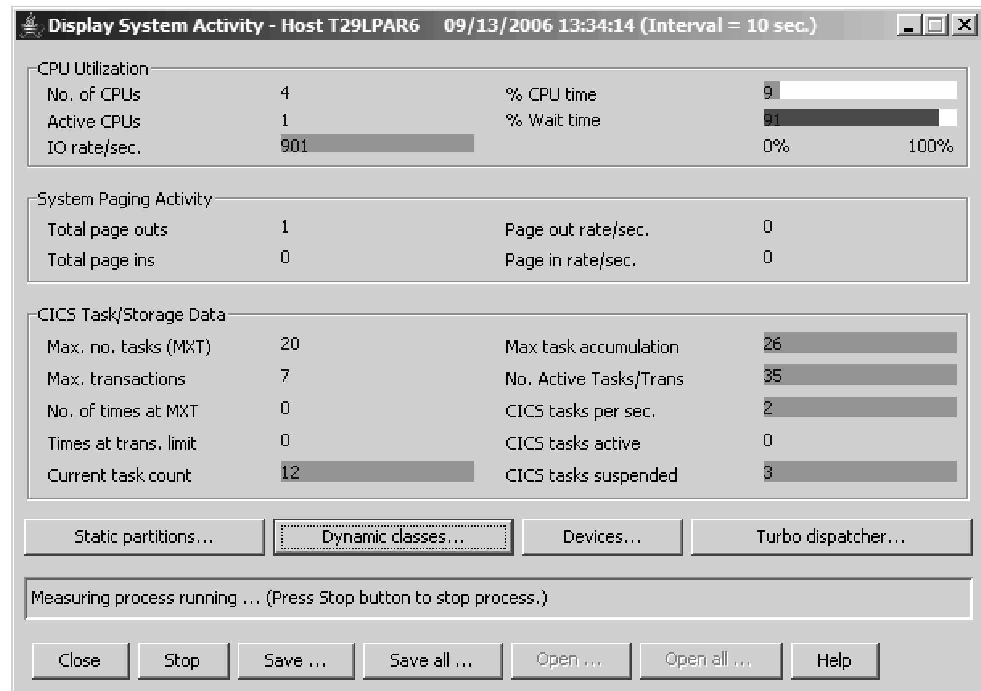


Figure 91. Displaying System Activity Using the VSE Navigator

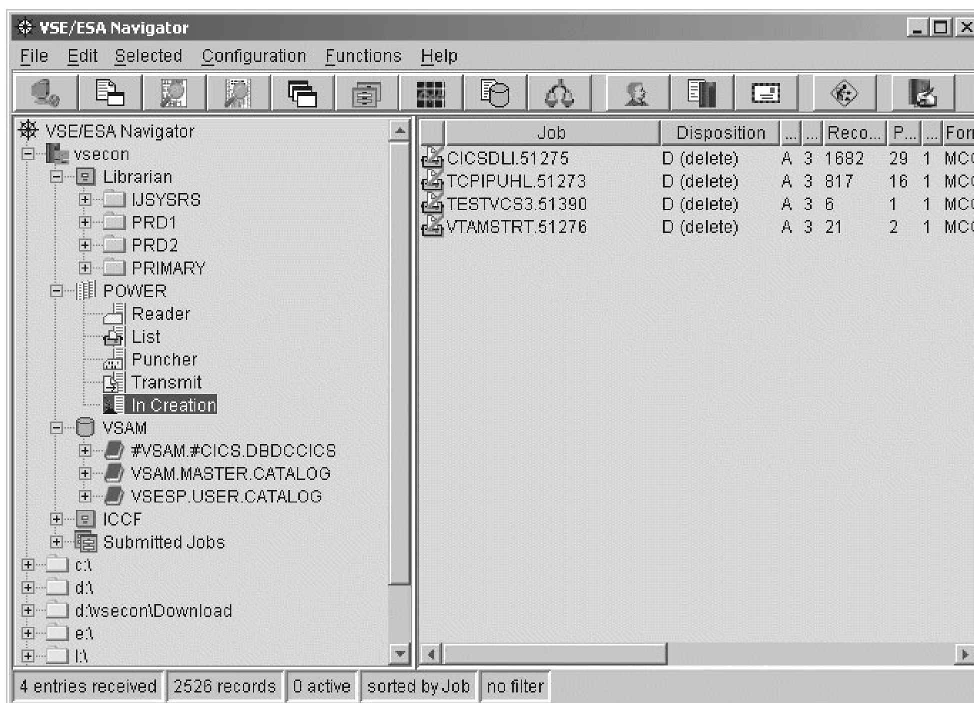


Figure 92. Displaying VSAM Files Using the VSE Navigator

Prerequisite for Using the VSE Navigator

Before installing and using the VSE Navigator, you should have installed / configured:

- The VSE Connector Client on your workstation, as described in "Installing the VSE Connector Client" on page 24.
- The VSE Connector Server on your host, as described in "Configuring the VSE Connector Server" on page 27.
- TCP/IP for VSE/ESA.
- The Java Development Kit (JDK) 1.5 or later. If you do not have JDK 1.5 or later installed, refer to "Installing and Configuring Java" on page 19 for details of how to install it.

Note: Also check the z/VSE Home Page for the latest APAR levels. The URL is:

`http://www.ibm.com/systems/z/os/zvse/`

Migrating From Earlier Versions

If you currently have an earlier version of the VSE Navigator installed, you must migrate to the latest version that runs with the current version of z/VSE. To do so, simply download the version of the VSE Navigator that runs with the current version of z/VSE, and use it to replace your existing version.

For details, see "Installing the VSE Navigator" on page 173 (below).

Installing the VSE Navigator

You install the VSE Navigator on a Java-enabled platform.

Note: Before you begin, ensure you have fulfilled all prerequisites described in “Prerequisite for Using the VSE Navigator” on page 172.

The VSE Navigator is supplied as one Java-installation class file, **setup.jar**. To install the VSE Navigator, you should:

1. Obtain the VSE Script Server from the Internet, by starting your Web browser and proceeding to URL:
   ```
   http://www.ibm.com/systems/z/os/zvse/downloads/
   ```
   From within the VSE Navigator section, select **Details and Download**. Then download the latest code to the directory where you wish to install the VSE Navigator, by selecting the file **vsenavi*nnn*.zip**. **Note:** *nnn* refers to the current VSE version (for example, **vsenavi430.zip**).
2. Unzip the file **vsenavi*nnn*.zip**, which contains these files:
   - setup.jar (contains the VSE Navigator code)
   - setup.bat (an install batch file for Windows XP/Vista/7)
   - setup.cmd (an install batch file for Windows NT)
   - setup.sh (an install script for Linux/Unix)
3. Start the batch file (by double-clicking the file) that is applicable to your operating-system platform.
4. The installation process now begins, and you are guided through various installation menus.

Starting the VSE Navigator Client

To start the VSE Navigator client:

On Operating System...
  The run file or shell script is ...

Windows XP/Vista/7
  run.bat

Windows NT
  run.cmd

Linux/Unix
  run.sh

On Windows you are provided with desktop objects you can use to start the VSE Navigator, access the online documentation (file **NaviReference.html**), and so on.

When starting the VSE Navigator client for the first time, you will be required to specify your local:

- utilities (for example, the Web browser you wish to use for displaying Help texts)
- directories

Adding Your Own VSE Navigator Plug-Ins

The VSE Navigator provides a Java programming interface that allows you to add your own "plug-in" functions to the VSE Navigator client. To write a VSE Navigator plug-in, you must:

1. Implement the methods of a Java interface.
2. Copy the class files into the VSE Navigator's plug-in directory. You may also create a new directory for your classes within the plug-in directory.
3. Restart the VSE Navigator client. Your plug-in is then dynamically loaded, and can be accessed from the toolbar and menus of the VSE Navigator. The online Programming Reference manual is HTML-based, and you can access it from **NaviReference.html**, which is stored in the directory where you installed the VSE Navigator.

Here is the GUI provided for using SSL with the VSE Navigator:

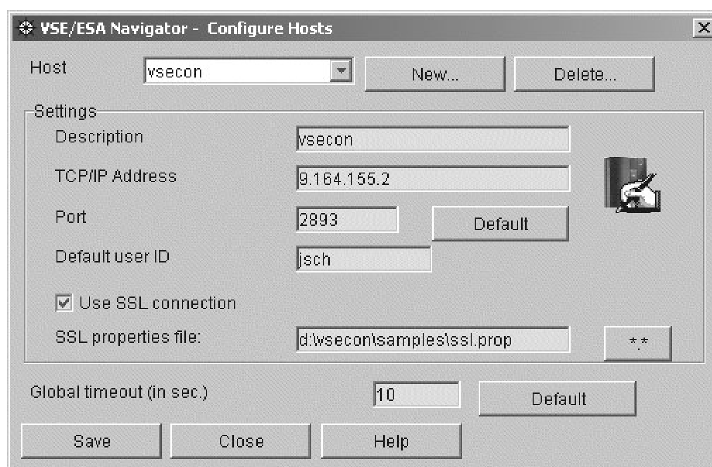


Figure 93. Configure Hosts for the VSE Navigator

Here is the GUI provided for using SSL with the VSE Navigator:

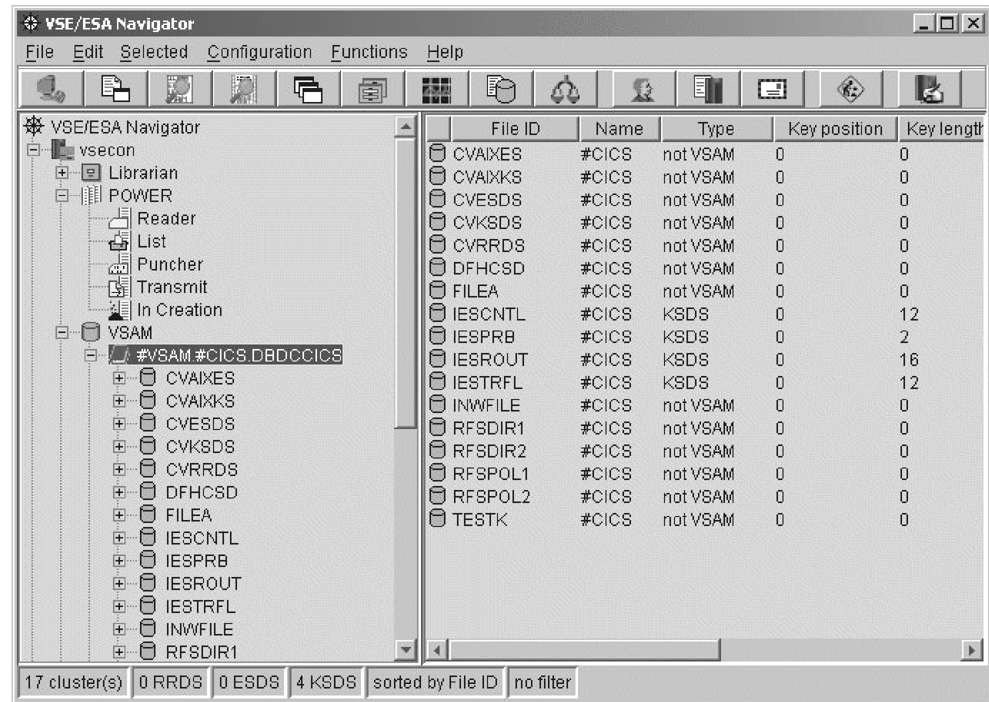


Figure 94. Using the VSE Navigator to Access CICS Data

Using the VSE Health Checker Application

The *VSE Health Checker* is a system diagnosis utility that you can use to retrieve, display, and analyze performance-related data from a VSE system.

The *client-part* of the VSE Health Checker, which communicates with the *VSE Connector Server*, provides you with a variety of functions. It can detect VSE configuration problems, which might result in decreased performance or even system outages. Gathered data can be exported and imported in XML format.

VSE data is *retrieved* by:
- Sending console commands.
- Submitting VSE/POWER jobs.
- Downloading VSE Librarian members.
- Invoking CICS transactions.

There is no dependency to any vendor tools.

The VSE output data is transferred to your workstation, parsed, and displayed in the GUI for further analysis.

Notes:
1. It is important to understand that the VSE Health Checker does **not** change any of the system parameters of your VSE system. All the actions that the VSE Health Checker performs are **read-only**.
2. Although the process of obtaining data from your VSE system might take up to several minutes, this is simply *elapsed time*. As a result, CPU overhead caused by the VSE Health Checker is minimal.

Here is the GUI provided by the VSE Health Checker:

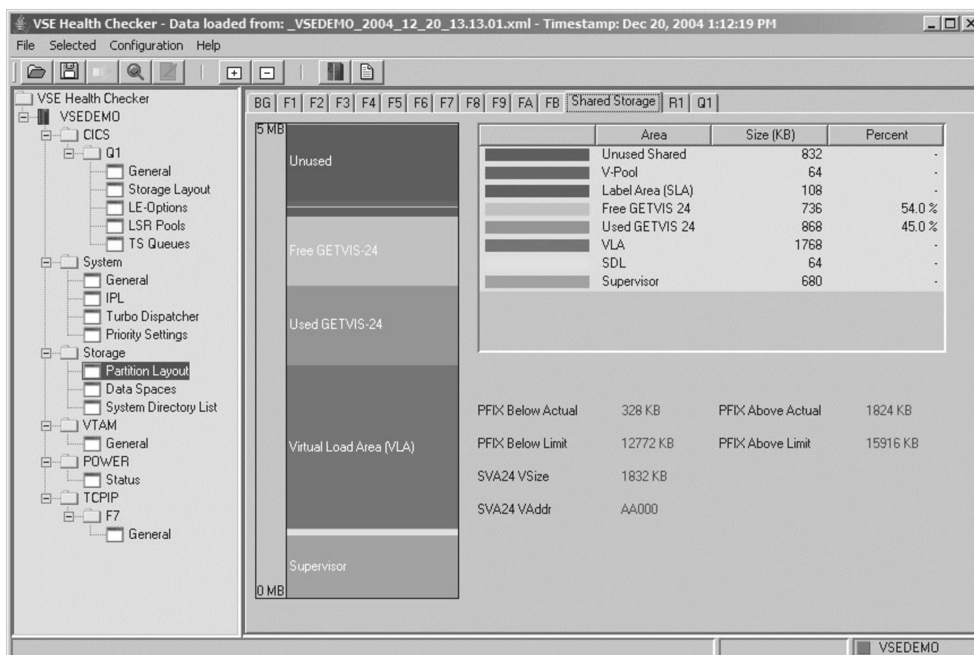


Figure 95. Graphical User Interface, as Provided by the VSE Health Checker

Prerequisite for Using the VSE Health Checker

Before installing and using the VSE Health Checker, you should have installed / configured:

- The VSE Connector Client on your workstation, as described in "Installing the VSE Connector Client" on page 24.
- The VSE Connector Server on your host, as described in "Configuring the VSE Connector Server" on page 27.
- TCP/IP for VSE/ESA.
- The Java Development Kit (JDK) 1.5 or later. If you do not have JDK 1.5 or later installed, refer to "Installing and Configuring Java" on page 19 for details of how to install it.

Note: Also check the z/VSE Home Page for the latest APAR levels. The URL is:

`http://www.ibm.com/systems/z/os/zvse/`

Migrating From Earlier Versions

If you currently have an earlier version of the VSE Health Checker installed, you must migrate to the latest version that runs with the current version of z/VSE. To do so, simply download the version of the VSE Health Checker that runs with the current version of z/VSE, and use it to replace your existing version.

For details, see "Installing the VSE Health Checker" on page 177 (below).

Installing the VSE Health Checker

You install the VSE Health Checker on a Java-enabled platform.

Note: Before starting, ensure you have fulfilled all prerequisites described in "Prerequisite for Using the VSE Health Checker" on page 176.

The VSE Health Checker is supplied as one Java-installation class file, **setup.jar**. To install the VSE Health Checker, you should:

1. Obtain the VSE Health Checker from the Internet, by starting your Web browser and proceeding to URL:

   ```
   http://www.ibm.com/systems/z/os/zvse/downloads/
   ```

   From within the VSE Health Checker section, download the latest code to the directory where you wish to install the VSE Health Checker, by selecting the file **vsehealth*nnn*.zip**. **Note:** *nnn* refers to the current VSE version (for example, **vsehealth430.zip**).
2. Unzip the file **vsehealth*nnn*.zip**, which contains these files:
   - setup.jar (contains the VSE Health Checker code)
   - setup.bat (an install batch file for Windows XP/Vista/7)
   - setup.cmd (an install batch file for Windows NT)
   - setup.sh (an install script for Linux/Unix)
3. Start the batch file (by double-clicking the file) that is applicable to your operating-system platform.
4. The installation process now begins, and you are guided through various installation menus.

Starting the VSE Health Checker Client

To start the VSE Health Checker client:

On Operating System...
  The run file or shell script is ...

Windows XP/Vista/7
  run.bat

Windows NT
  run.cmd

Linux/Unix
  run.sh

On Windows you are provided with desktop objects you can use to start the VSE Health Checker, access the online documentation (file **vsehealth.html**), and so on.

When starting the VSE Health Checker client for the first time, you will be required to specify your local Web browser you wish to use for displaying Help texts. For a description of how you to get started using the VSE Health Checker, refer to the topic "Quick Start" in the online documentation that is provided with the VSE Health Checker.

Chapter 16. Using JDBC to Access VSAM Data

This chapter describes how you can set up and issue relational database queries and update requests against VSAM data using a *Java Database Connectivity* (JDBC) driver. To do so, the Java-based connector provides various classes for use with JDBC.

You can therefore use SQL constructs to access VSAM data, instead of coding against the VSE Java Beans interface. Currently, only a subset of the SQL syntax is supported (shown in "SQL Statements That Are Supported by JDBC"). However, this is the same subset that can be used with the VSAM SQL call level interface (CLI) from within a DB2-Stored Procedure (see "Using DB2 Stored Procedures to Access VSAM Data" on page 395 for details).

The advantages of using the JDBC driver instead of the VSE Java Beans interface are:

- JDBC and SQL are standard interfaces for accessing data in relational databases.
- Many products such as the *IBM Visual Age for Java* support the JDBC interface.
- You can integrate VSAM access into applications that were created using the *IBM Visual Age for Java* program, without needing to include VSE-specific code.

For details of how to map non-relational VSAM data to a relational structure, refer to Chapter 12, "Mapping VSE/VSAM Data to a Relational Structure," on page 103.

This chapter contains these main topics:

- "SQL Statements That Are Supported by JDBC"
- "Relational and VSE Java Beans Terminology" on page 182
- "Example of Using JDBC to Access VSAM Data" on page 182

SQL Statements That Are Supported by JDBC

These are the SQL statements that are supported by JDBC:

Table 5. SQL Statements Supported by JDBC

<table>
<tr><th colspan="2">SQL Statement</th></tr>
<tr><td><insert statement> =</td><td>"INSERT" ["INTO"] <table name>
[<column name list>]
"VALUES"
"(" <insert values list> ")"
("," "(" <insert values list> ")")*
|
<select statement></td></tr>
<tr><td><table name> =</td><td><IDENTIFIER>
"\" <IDENTIFIER>
"\" <IDENTIFIER>
["\" <IDENTIFIER>]</td></tr>
<tr><td><column name list> =</td><td>"(" <column name>
("," <column name>)* ")"</td></tr>
<tr><td><column name> =</td><td><IDENTIFIER></td></tr>
<tr><td><insert values list>=</td><td>"(" <insert value>
("," <insert values>)* ")"</td></tr>
</table>

© Copyright IBM Corp. 2000, 2013

Table 5. SQL Statements Supported by JDBC (continued)

<table>
<tr><th colspan="2">SQL Statement</th></tr>
<tr><td><insert value> =</td><td><NUMBER>
| <CHAR LITERAL>

| <PREPARED EXPR></td></tr>
<tr><td><update statement> =</td><td>"UPDATE" <table name>
"SET" <column values>
[<where clause>]</td></tr>
<tr><td><column values> =</td><td><column name> "=" <updated value>
("," <column name> "=" <updated value>)*</td></tr>
<tr><td><updated value> =</td><td><NUMBER>
| <CHAR LITERAL>

| <PREPARED EXPR></td></tr>
<tr><td><delete statement> =</td><td>"DELETE" ["FROM"] <table name>
[<where clause>]</td></tr>
<tr><td><query statement> =</td><td><select statement></td></tr>
<tr><td><select statement> =</td><td><select no order>
[<order by clause>]</td></tr>
<tr><td><select no order> =</td><td>"SELECT" ["ALL" | "DISTINCT"]
<select list>
<from clause>
[<where clause>]</td></tr>
<tr><td><select list> =</td><td>"*"
|
<select item> ("," <select item>)*</td></tr>
<tr><td><select item> =</td><td>(<IDENTIFIER> "\"
    [<IDENTIFIER> "\" <IDENTIFIER> "\"
        [<IDENTIFIER> "\"]]
    "*"
)
|
(
    <sl element>
    ["AS"]
    [<IDENTIFIER> <QUOTED IDENTIFIER>]
)</td></tr>
<tr><td><sl element> =</td><td><sl mult expr>
(("+" | "-") <sl mult expr>)*</td></tr>
<tr><td><sl mult expr> =</td><td><sl primary expr>
(("*" | "/") <sl primary expr>)*</td></tr>
<tr><td><sl primary expr> =</td><td><table column>
| <NUMBER>

| <CHAR LITERAL>

| <PREPARED EXPR>

| "(" <sl element> ")"</td></tr>
<tr><td><from clause> =</td><td>"FROM" <from item>
[
    ("," <from item>)+
    | (<join clause>)+
]</td></tr>
</table>

Table 5. SQL Statements Supported by JDBC (continued)

<table>
<tr><th colspan="2">SQL Statement</th></tr>
<tr><td><from item> =</td><td>(
"(" <subquery> ")"
|
<table name>
)
["AS"] [<IDENTIFIER>]</td></tr>
<tr><td><join clause> =</td><td><join type> <from item> [<join specification>]</td></tr>
<tr><td><join type> =</td><td>"INNER" "JOIN"
| "NATURAL" ["INNER"] "JOIN"

| "LEFT" ["OUTER"] "JOIN"

| "RIGHT" ["OUTER"] "JOIN"

| "FULL" ["OUTER"] "JOIN"

| "JOIN"</td></tr>
<tr><td><join specification> =</td><td>"USING" "(" <IDENTIFIER>
("," <IDENTIFIER>)* ")"
| "ON" <table column> "=" <table column></td></tr>
<tr><td><where clause> =</td><td>"WHERE" <where expr></td></tr>
<tr><td><where expr> =</td><td><where and expr>
("OR" <where and expr>)*</td></tr>
<tr><td><where and expr> =</td><td><where rel expr>
("AND" <where rel expr>)*</td></tr>
<tr><td><where rel expr> =</td><td>["NOT"]
(
<where primary expr>
<relop>
<where primary expr>
)
| "(" <where expr> ")"</td></tr>
<tr><td><where primary expr> =</td><td><table column>
| <NUMBER>

| <CHAR_LITERAL>

| <PREPARED EXPR></td></tr>
<tr><td><order by clause> =</td><td>"ORDER" "BY"
<table column> ["ASC" | "DESC"]
("," <table column> ["ASC" | "DESC"])*</td></tr>
<tr><td><relop> =</td><td>"="
| "<>"

| "<"

| "<="

| ">"

| ">="</td></tr>
<tr><td><subquery> =</td><td><select no order></td></tr>
<tr><td><table column> =</td><td><IDENTIFIER>
["\" <IDENTIFIER>
["\" <IDENTIFIER> "\" <IDENTIFIER>
["\" <IDENTIFIER>]]]</td></tr>
</table>

Relational and VSE Java Beans Terminology

Here are some guidelines for understanding how terms that are used with relational SQL, correspond to non-relational VSE Java Beans.

Table 6. Relational Terms and Their VSE Equivalents

SQL term	VSE Java Beans term
SQL table	A VSEVSAMCatalog, a VSEVsamCluster, together with a VSEVsamMap (and optionally VSEVsamView)
Database row	VSEVsamRecord, together with a VSEVsamMap describing the column names and data field properties.
Column name	VSEVsamField, describing the column's: Name Offset within the record Length Data type (string, integer, and so on)

Specifying Table Names

The following example illustrates how you specify table names with JDBC. Assume the cluster `MY.TEST.CLUSTER` resides in catalog `MY.USER.CATALOG`. A map has been defined for this cluster with the name `MY.TEST.MAP`.

The table name used for the VSAM JDBC driver will therefore be:

```
MY.USER.CATALOG\MY.TEST.CLUSTER\MY.TEST.MAP
```

Example of Using JDBC to Access VSAM Data

This topic provides a detailed example of how you can access VSAM data using the JDBC driver.

The example performs roughly the same processing as the *FlightOrderingServlet* described in “Example of How to Implement a Servlet” on page 230. It operates on the same VSAM clusters as the *FlightOrderingServlet*, but does not create these clusters. Therefore, if you have not already created these clusters (and filled them with data), refer to “Creating the VSAM Clusters for the Sample” on page 231 for details of how to do so.

The example servlet is implemented in the Java source file **JdbcFlightOrderingServlet.Java**, which is supplied with the online documentation (see “Using the Online Documentation Options” on page 26 for details).

The example begins by specifying the class name:

```
public class JdbcExample
{
```

The main steps for the rest of the example are shown below.

Step 1. Define the Local Variables

In this step, the local variables are defined. This example assumes that these VSAM files have already been defined and filled with sample data:
- FLIGHT.ORDERING.FLIGHTS (KSDS)
- FLIGHTS.ORDERING.ORDERS (RRDS)

(for details of how to do so, see "Creating the VSAM Clusters for the Sample" on page 231).

```
String vsamCatalog    = "VSESP.USER.CATALOG";
String flightsCluster = "FLIGHT.ORDERING.FLIGHTS";
String ordersCluster  = "FLIGHT.ORDERING.ORDERS";
String flightsMapName = "FLIGHTS_MAP";
String ordersMapName  = "ORDERS_MAP";

public static void main(String argv[]) throws IOException
{
   try
   {
```

Figure 96. VSAM Data via JDBC: Define Local Variables

Step 2: Prompt for IP address, User ID, and Password

In this step, the user is prompted to enter the IP address of the z/VSE host system, User ID, and password.

```
BufferedReader r = new BufferedReader(
                      new InputStreamReader(System.in));
System.out.println("Please enter your VSE IP address:");
String ipAddr = r.readLine();
System.out.println("Please enter your VSE user ID:");
String userID = r.readLine();
System.out.println("Please enter password:");
String password = r.readLine();
```

Figure 97. VSAM Data via JDBC: Prompt for IP Address, User ID, Password

Step 3. Establish a Connection to the z/VSE Host

In this step, an instance of the VSAM JDBC driver is created and a z/VSE host connection is established.

```
java.sql.Connection jdbcCon;
   java.sql.Driver jdbcDriver = (java.sql.Driver) Class.forName(
           "com.ibm.vse.jdbc.VsamJdbcDriver").newInstance();

   // Build the URL to use to connect
   String url = "jdbc:vsam:"+ipAddr;

   // Assign properties for the driver
   java.util.Properties prop = new java.util.Properties();
   prop.put("port", 2893);
   prop.put("user", userID);
   prop.put("password", password);

   // Connect to the driver
   jdbcCon = DriverManager.getConnection(url, prop);
}
catch (Throwable t)
{
 ⋮
}
```

Figure 98. VSAM Data via JDBC: Establish a Host Connection

Step 4. Display a List of Rows in the Database

In this step, an SQL statement is created to display a list of database rows (which are VSAM records).

```
try
{
   // Get a statement
   java.sql.Statement stmt = jdbcCon.createStatement();

   // Execute the query ...
   java.sql.ResultSet rs = stmt.executeQuery(
      "SELECT * FROM "+vsamCatalog+"\\"+flightsCluster+"\\"+flightsMapName);
```

Figure 99. VSAM Data via JDBC: Display the Database Rows

Step 5. Process Result-Set Returned From JDBC

```
   while (rs.next())
   {
      int flightNumber   = rs.getInt("FLIGHT_NUMBER");
      String start       = rs.getString("START");
      String destination = rs.getString("DESTINATION");
      String departure   = rs.getString("DEPARTURE");
      String arrival     = rs.getString("ARRIVAL");
      int price          = rs.getInt("PRICE");
      String airline     = rs.getString("AIRLINE");
   }
   rs.close();
   stmt.close();
}
catch (SQLException t)
{
 ⋮
}
```

Figure 100. VSAM Data via JDBC: Process Result-Set

Step 6. Add a New Record

In this step, a new record is added to the VSAM database.

```
try {
   java.sql.PreparedStatement pstmt = jdbcCon.prepareStatement(
      "INSERT INTO "+vsamCatalog+"\\"+flightsCluster+"\\"+flightsMapName+
      " (FLIGHT_NUMBER,START,DESTINATION,DEPARTURE,ARRIVAL,PRICE,AIRLINE)"+
      " VALUES(?,?,?,?,?,?,?)");

   pstmt.setInt(1, 398);
   pstmt.setString(2,"Honolulu");
   pstmt.setString(3,"Bankok");
   pstmt.setString(4,"07:30");
   pstmt.setString(5,"22:45");
   pstmt.setInt(6, 1500);
   pstmt.setString(7,"VSE Airtours");

   // Execute the query
   int num = pstmt.executeUpdate();
   pstmt.close();
}
catch (SQLException t)
{
.
.
.
}
}
}
```

Figure 101. VSAM Data via JDBC: Add a New Record

Chapter 17. Using Java Applets to Access Data

Java applets are Java programs that run inside the Java Virtual Machine of a Web browser. The main advantages of using applets are that:
- They can be accessed using any Java-enabled Web browser.
- There is no need to install any further programs on the Web client.
- They allow you to build an easy-to-use User Interface (UI).

Applets are by definition *secure*. They cannot access:
- any of the resources of the client workstation
- the client workstation's memory
- the network.

Here is an example of an applet tag:

```
<applet code="myapplet" archive="applets.jar">
</applet>
```

The "applet code" tag specifies the name of the main Java class of the applet, the optional "archive" tag specifies one or more archives where the class library (and all other required classes) are located. You can store the class and JAR files belonging to the applet in any directory of the physical/logical middle-tier.

This chapter contains these main topics:

- "How Applets Are Used in 2-Tier Environments"
- "How Applets Are Used in 3-Tier Environments" on page 189
- "How the VSEAppletServer Is Used" on page 191
- "Disadvantages and Restrictions Of Using Applets" on page 191
- "Running the Sample Data-Mapping Applet" on page 192
- "Running the Sample VSAM Applet" on page 201
- "Running the Sample DL/I Applet" on page 214

How Applets Are Used in 2-Tier Environments

Figure 102 on page 188 shows how an applet is used within the z/VSE *2-tier environment*:

© Copyright IBM Corp. 2000, 2013

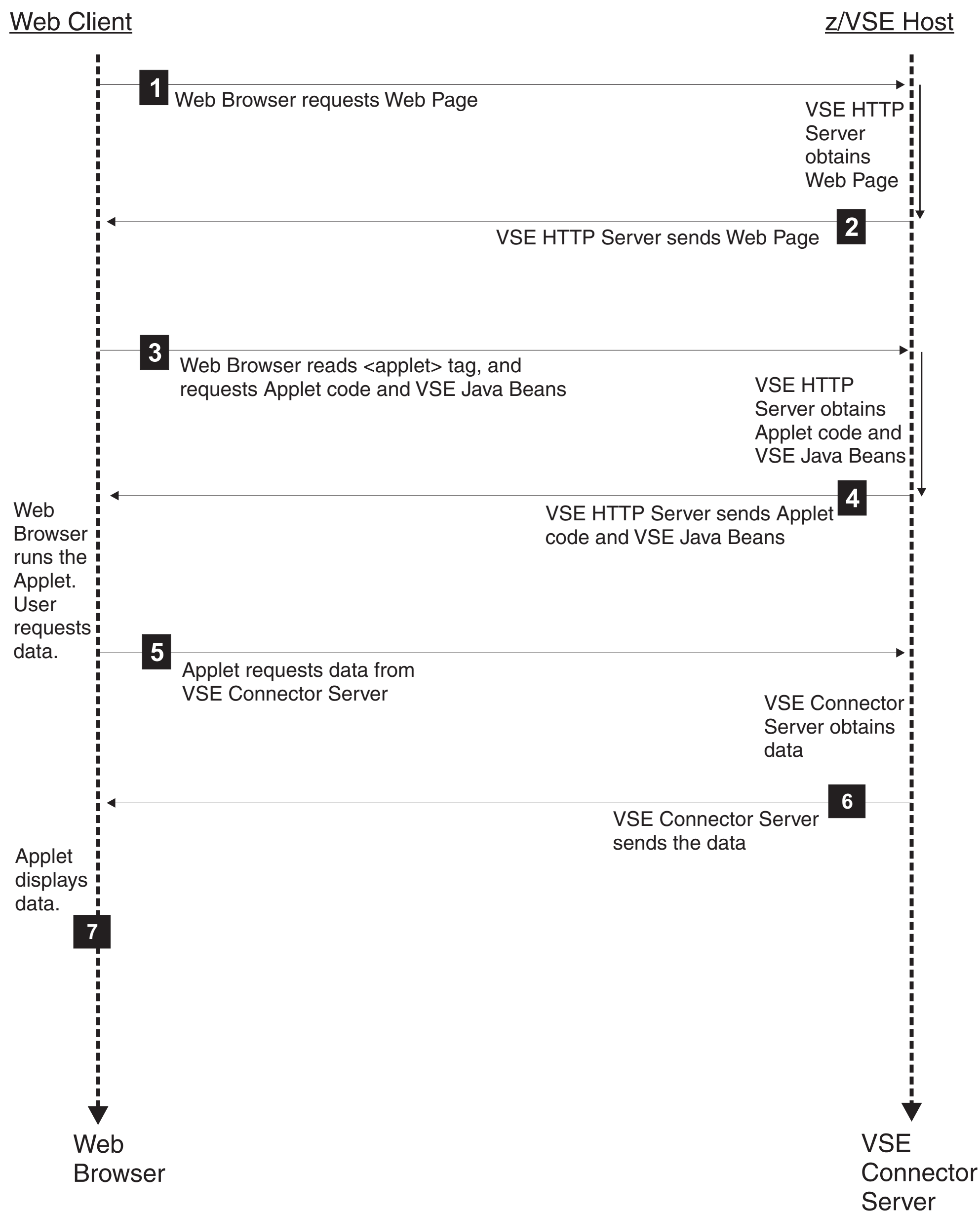


Figure 102. How Applets Are Used in the z/VSE 2-Tier Environment

HTTP Sessions are used between the Web Client and the z/VSE host for sending and receiving data.

The number of each list item below describes a step shown in Figure 102:

1 The client's Web browser requests an HTML page from the VSE HTTP Server running on the z/VSE host.

2 The VSE HTTP Server sends the Web Page to the client's Web browser.

3 The client's Web browser reads an <applet> tag and requests the applet code from the VSE HTTP Server on the z/VSE host. The applet code is stored in one or more JAR files. The Web browser also requests the VSE Java Beans class library (**VSEConnector.jar**) from the VSE HTTP Server.

4 The VSE HTTP Server sends the applet code to the client's Web browser, together with the VSE Java Beans class library.

5 The client's Web browser runs the applet. The applet uses the VSE Java Beans class library (**VSEConnector.jar**) to build a connection to the VSE Connector Server. The end-user requests data that is stored on the z/VSE host. For an applet in a 2-tier environment, the data can be VSE/POWER, VSE/VSAM, VSE/ICCF, or Librarian data.

Note: An applet *cannot* obtain DB2, DL/I, or CICS data in a 2-tier environment. This is because the VSE Connector Server cannot access these systems. In addition, the use of WebSphere MQ Servers is not possible in 2-tier environments.

6 The VSE Connector Server obtains the required data using "native" calls (using the standard access methods), and then sends the data to the client's Web browser via TCP/IP.

7 The client's Web browser runs the applet a second time, and displays the Web Page together with the requested data.

Although applets are mainly used in 2-tier environments, you can also use an applet within the *3-tier environment* providing you implement a "router" on the physical/logical middle-tier, that serves as a gateway between the client and the z/VSE host. For details, see "How the VSEAppletServer Is Used" on page 191.

How Applets Are Used in 3-Tier Environments

Figure 103 on page 190 shows how an applet is used within the z/VSE *3-tier environment*. The number of each list item below describes a step shown in this figure.

1 The client's Web browser requests an HTML page from the IBM HTTP Server running on the physical/logical middle-tier.

2 The IBM HTTP Server sends the Web Page to the client's Web browser.

3 The client's Web browser reads an <applet> tag and requests the applet code from the IBM HTTP Server running on the physical/logical middle-tier. You can store the class and JAR files belonging to the applet in any directory of the physical/logical middle-tier. The Web browser also requests the VSE Java Beans class library (**VSEConnector.jar**) from the IBM HTTP Server.

4 The IBM HTTP Server sends the applet code to the client's Web browser, together with the VSE Java Beans class library.

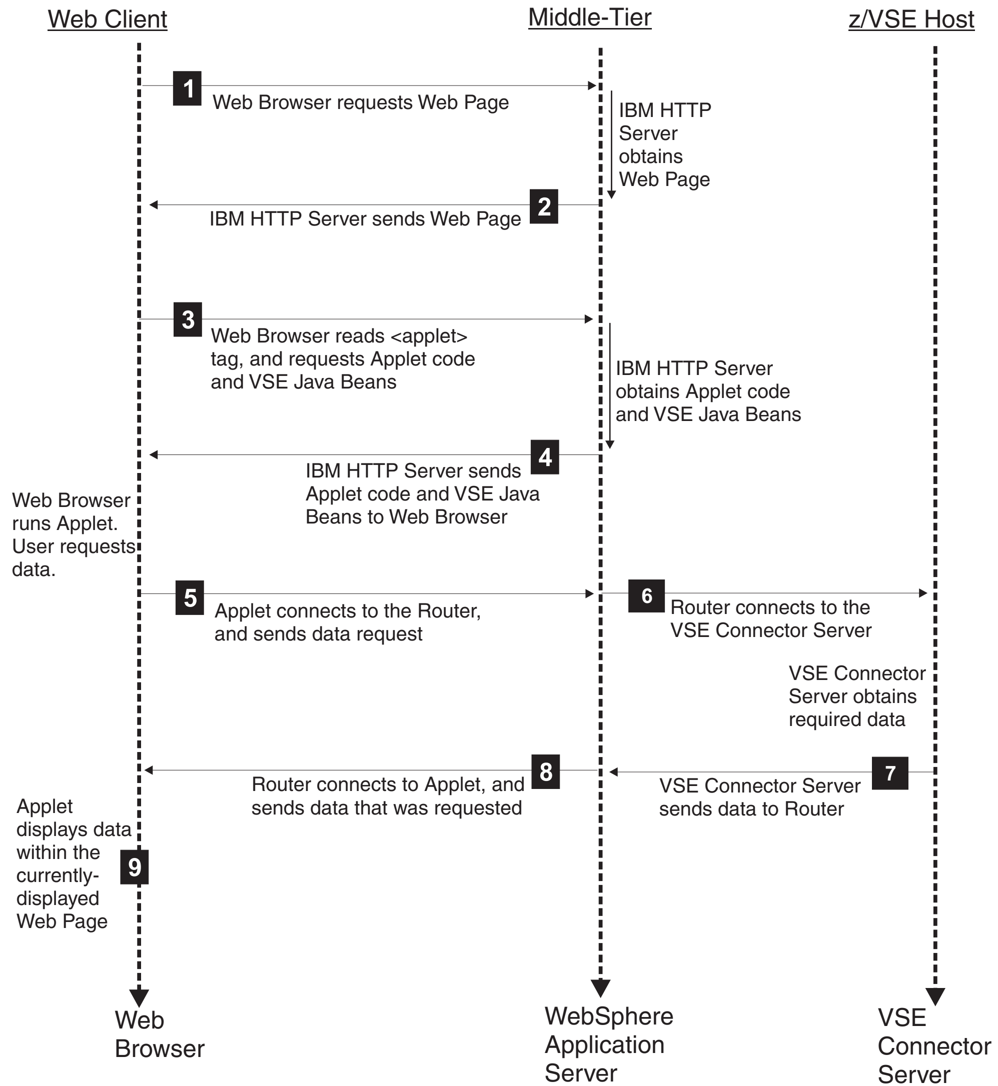


Figure 103. How Applets Are Used in the z/VSE 3-Tier Environment

5 The client's Web browser runs the applet, and the end-user requests data that is stored on the z/VSE host. The applet connects to the router on the physical/logical middle-tier, and sends the request for data to the router. The router is the *VSEAppletServer* (described in "How the VSEAppletServer Is Used" on page 191) running on the physical/logical middle-tier.

6 The router connects to the VSE Connector Server running on the z/VSE host, and forwards the request for data to it.

7 The VSE Connector Server retrieves the data using "native" calls (the standard access method), and sends the data back to the *VSEAppletServer* router running on the physical/logical middle-tier (via TCP/IP).

Notes:

1. The VSE Connector Server can be used for accessing VSE/VSAM, VSE/POWER, VSE/ICCF, or Librarian data.
2. An alternate method for accessing VSAM data stored on the z/VSE host, is to use a DB2 Stored Procedure on the physical/logical

middle-tier which communicates directly with the VSAM file system on the z/VSE host. This is described in "Using DB2 Stored Procedures to Access VSAM Data" on page 395.

8 The router connects to the applet running in the client's Web browser, and sends the data to the applet (also via TCP/IP).

9 The applet running inside the client's Web browser displays the data within the currently-displayed Web Page.

HTTP Sessions are used between the Web Client and the physical/logical middle-tier for sending and receiving data. Connect Sessions are used between the physical/logical middle-tier and the z/VSE host for sending and receiving data.

How the VSEAppletServer Is Used

Applets, by nature, have many restrictions, including:

- They can open a new network connection only to the platform from which they are downloaded. In a 3-tier environment this is the physical/logical middle-tier.
- They can only access the file system of the platform from which they are downloaded. In a 3-tier environment this is a file system stored on the physical/logical middle-tier.

To get around these problems and allow applets to get data from the VSE Connector Server running on the z/VSE host, a simple router (*VSEAppletServer*) is provided. The applet simply connects to this router. The router, which does not have this restriction, then connects to the VSE Connector Server to get VSE-based data, and pass it back to the applet.

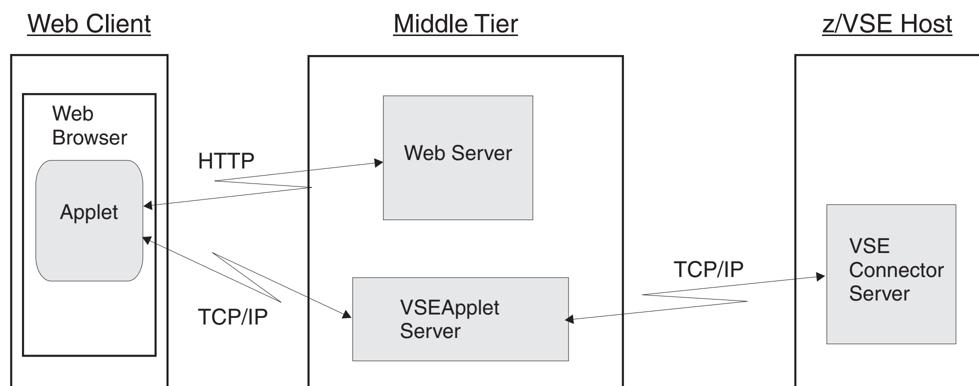


Figure 104. How the VSEApplet Server Is Used in the 3-Tier Environment

Disadvantages and Restrictions Of Using Applets

These are the disadvantages and restrictions when using applets in the z/VSE environment:

- Since the applet must run in a Web browser, you do not have the advantage of having an extremely fast physical/logical middle-tier machine of the 3-tier environment. The speed of the process is instead dependent on the resources of each individual Web browser, and the network speed and bandwidth.
- You cannot use the WebSphere MQ Client for Java as the connector for an applet, because the WebSphere MQ Server for z/VSE can only communicate with another WebSphere MQ Server, and not directly with the WebSphere MQ Client for Java.

- You can only use the **archive** tag with Netscape, or with the Microsoft Internet Explorer 4 (or later), not with the Microsoft Internet Explorer 3. Check the documentation belonging to your Web browser for details about supported applet parameters.
- You must store class and jar files as *binary* in the VSE library system. You can use the Librarian **LD** command to check the file format.
- When you write *applets*, you should ***never*** "hard-code" any user IDs and passwords in the applet code: when the applet is downloaded to a Web browser and is stored in the Web browser's cache, this information could possibly be displayed by unauthorized persons.

Running the Sample Data-Mapping Applet

The information in this topic is based upon a sample applet, the *VSAM data-mapping applet* (also referred to as simply the *data-mapping applet*) that is provided with the VSE Connector Client.

Other examples of applets are similarly provided with the VSE Connector Client, such as the:

- *VsamSpaceUsage* (which displays the used and free VSAM space).
- *DB2ConnectorJDBCApplet* (which calls a DB2 Stored Procedure to access VSAM data via the VSAMSQL *Call Level Interface*, abbreviated to *CLI*). For details, see "Running the Sample VSAM Applet" on page 201.
- *VSAM Applet* (described on page "Running the Sample VSAM Applet" on page 201).
- *DL/I Applet* (described on page "Running the Sample DL/I Applet" on page 214).

Description of the Data-Mapping Applet

The data-mapping applet describes a Java applet that allows you to create and maintain maps and views for specific VSAM files. It does not show you how to display or modify VSAM data itself. To display or modify VSAM data, you must either use:

- an applet with the same functionality as the sample VSAM applet (described in "Example of How to Implement a Servlet" on page 230).
- a servlet (described in "Example of How to Implement a Servlet" on page 230).

Note: The data-mapping applet *might not run* with certain combinations of operating system and Web browser.

The applet performs this general processing:

1. The applet displays some window controls, which allow the user to enter the IP address of a z/VSE host, a VSE user ID, and a password.
2. The applet connects to the VSE Connector Server, and retrieves a list of VSAM catalogs.
3. By double-clicking on an item in the catalog list, a list with all clusters in this catalog is displayed.
4. By double-clicking on a cluster, a list showing all maps defined to this cluster is displayed.

5. By double-clicking on a map, two lists are displayed. The first list shows all data fields of this map, the second list shows all views of this map.
6. By double-clicking on a view, a list with all fields of this view is displayed.

Figure 105 shows the applet running in the Web browser window.

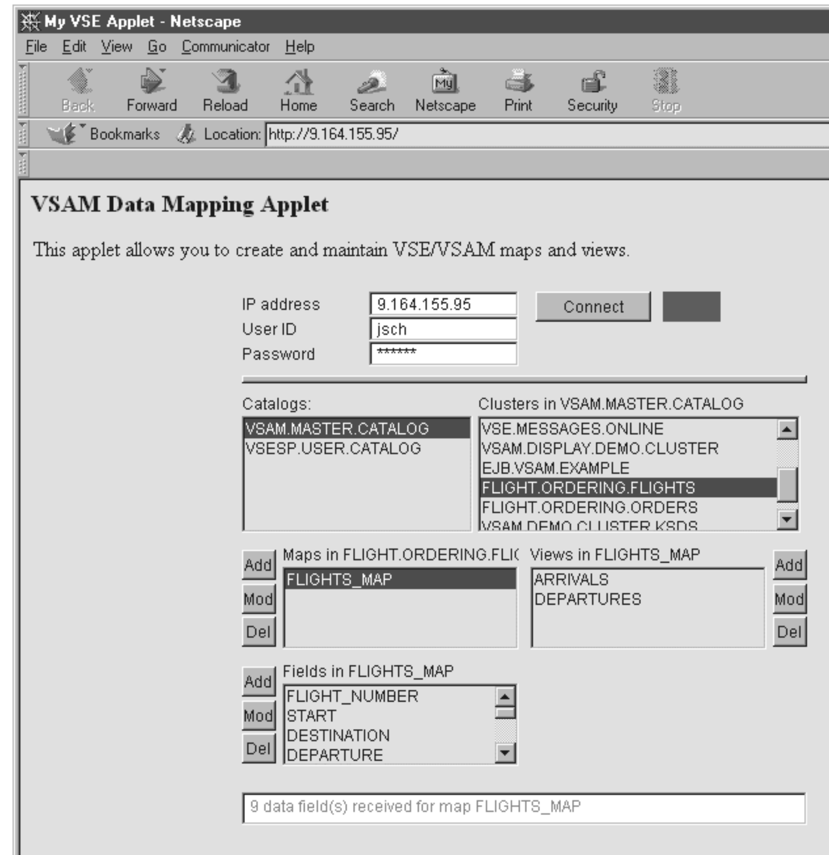


Figure 105. Window for VSAM Data-Mapping Applet

The VSAM file FLIGHT.ORDERING.FLIGHTS contains data records describing flights. Each flight consists of data fields (DEPARTURE, ARRIVALS, SEATS, and so on). These data fields are contained in the map FLIGHTS_MAP that describes the complete record.

Using push-buttons that are displayed next to the lists shown in Figure 105, you can add, change, or delete, the maps, views, and fields. By clicking a push button to add or modify a map, view, or field, a dialog is displayed that allows you to enter new data.

The map has two different data views, DEPARTURES and ARRIVALS, that provide subsets of the data fields of the map. All maps and views are stored in a VSAM file which contains the VSAM mapping definitions. For details, see "How Maps Are Stored on the z/VSE host" on page 104.

Activities Required on the z/VSE Host

Before you can run the VSAM data-mapping applet, you must perform these activities on your z/VSE host:

1. Define a Web server on z/VSE, by entering this TCP/IP command on the z/VSE console:

   ```
   DEFINE HTTPD,ID=MYHTTPD,ROOT=PRIMARY.TEST
   ```

   This will start an HTTP daemon with root library PRIMARY.TEST.
2. Create a file with the name **index.html**, that will be read when a Web browser connects to the z/VSE host. Here is the **index.html** file you should use with the sample applet:

   ```
   <html>
   <head>
   <title>VSAM Data Mapping Example Applet</title>
   </head>
   <body>
   <h2>VSAM Data Mapping Applet</h2>
   This applet can be used to create and maintain VSAM maps and views.
   Please logon to your VSE host. When the connection is established, a
   list of VSAM catalogs is displayed.
   <p>
   <center>
   <applet code="com.ibm.vse.samples.VsamMappingApplet" width=440 height=420
           archive="applets.jar, vsecon.jar">   1
   </applet>
   </center>
   </body>
   </html>
   ```

   1 The archive tag specifies the names of the Java archives (.JAR) containing all classes that are required to run the applet. In this example:
   - File **applets.jar** contains the applet-specific code.
   - File **vsecon.jar** is identical to **VSEConnectors.jar**, except that it has been given a short file name so it can be placed in a VSE library.
3. Place file **index.html** in the HTTP server's *root* library.

Deploying the Data-Mapping Applet

To deploy the data-mapping applet, you must:

1. From the **vsecon\samples** directory, compile the Java sources, and create the JAR archive. To do so, use the following statements:

   ```
   call javac com\ibm\vse\samples\VsamMappingApplet.java
   call javac com\ibm\vse\samples\VsamAppletListener.java
   call jar c0fv applets.jar com\ibm\vse\samples\VsamMappingApplet.class
                             com\ibm\vse\samples\VsamAppletListener.class
   ```

2. Send the created JAR archive to the VSE HTTP server's root directory (in *binary* format). To do so, you can either use an emulator program, or FTP. The JAR utility is part of your local Java installation. For details of how to use FTP, see page "Obtaining a Copy of the VSE Connector Client" on page 25.

Calling the Data-Mapping Applet

To run the data-mapping applet (or any other applet), it must be:

1. Downloaded from the z/VSE host to a local workstation.
2. Executed in the Java Virtual Machine of a Web browser installed on the local workstation.

If the VSE HTTP Server is running, to display the above HTML file, you simply enter the IP address or symbolic name of your z/VSE host in the Web browser's *address/location* field. To start the VSE HTTP Server, you enter this TCP/IP command at the z/VSE console:

```
xx q httpds
```

where *xx* is the reply-ID of the TCP/IP partition.

How Various Web Browsers Search for JAR and Class Files

- The *Netscape Communicator* does not search in the local classpath. Instead, it always takes the classes from the path specified in the applet's archive tag.
- The *Microsoft Internet Explorer 3* cannot process the archive tag.
- The *Microsoft Internet Explorer 4 and 5* place your local system classpath in front of the path of JAR files that is specified in the applet's archive tag. As a result, if you have same classes locally, the z/VSE host-based classes will not be loaded!. The Web browser will instead load the *local* classes.

Setting Up the Data-Mapping Applet Class

In general, an applet extends the Java applet class (supplied with the Java Development Kit). The provided data-mapping applet however, also implements an *ActionListener* that detects push-button actions.

```
/* Import applet classes */
import java.applet.*;
...

public class VsamMappingApplet extends Applet
implements ActionListener, ItemListener
{
   VSESystem system;
   VSEConnectionSpec spec;
   VSEVsam vsam;
   VSEVsamCatalog catalog;
   VSEVsamCluster cluster;
   VSEVsamMap map;
   VSEVsamView view;
   VSEVsamField field, newField;
   VsamAppletListener vl;
   ...
```

Figure 106. Data-Mapping Applet Code for Setting Up the Java Class

Initializing the Data-Mapping Applet

The *init()* method is called from the Web browser when the applet is first started. In this example:

1. The logon controls are displayed.
2. A frame is created that is required as the parent frame of the various dialogs for adding, modifying, or deleting maps, views, and data fields.

```
public void init()
{
   f = new Frame();
   pgl = new PowerGridLayout(100, 66); // a box with 100 x 66 units   1
   setLayout(pgl);
   vMapFields = new Vector();
   vViewFields = new Vector();
   displayLogonDialog();       // show the dialogbox
   repaint();
}
```

Figure 107. Sample Code for Initializing the Data-Mapping Applet

1 The *PowerGridLayout* class is a Java layout manager that is much used in the *VSE Navigator* function, and some examples contained in the VSE Connector Client. The *PowerGridLayout* class is contained in the **com.ibm.vse.utilities** package.

Re-Displaying or Leaving an HTML Page

The applet *start()* method is called from the Web browser whenever the HTML page is to be re-displayed:

```
public void start()
{
}
```

The applet *stop()* method is called from the Web browser when leaving the HTML page:

```
public void stop()
{
}
```

Using the Data-Mapping Applet to Add a Map to a VSAM Cluster

This code shows how to define a *VSEVsamMap* using the data-mapping applet. The same basic method is also used for defining views and fields:

1. A local map object is created using its constructor.
2. The local map object is then created on the z/VSE host, using the *create()* method.

Accessing the z/VSE host can produce an *IOException* or a *ConnectorException* error. Therefore this call must be contained in a “try-catch” clause.

```
public int addMap()
{
   ...

   /* Get map name from textfield on dialogbox */
   String name = tfName.getText().toUpperCase();

   /* Create local object */
   map = new VSEVsamMap(   1
           system,
           ((VSEVsamCatalog)(vCatalogs.elementAt(catIndex))).getFileID(),
           ((VSEVsamCluster)(vClusters.elementAt(cluIndex))).getFileID(),
           name);

   /* Create map on host */
   try {
      map.create();   2

      /* Add new mapname to map list ... */
      ...
   }
   catch (Exception e)
   {
      ...
   }
   ...
}
```

Figure 108. Data-Mapping Applet Code for Adding a Map to a VSAM Cluster

The number below refer to the numbers in Figure 108:

1 Here a new local instance of class *VSEVsamMap* is created. At this stage, no host action has been performed.

2 The *create()* method is used to create the map definition on the host.

Using the Data-Mapping Applet to Modify a Map

A map object can be renamed using the *rename()* method, which sends a request to the z/VSE host to change the map's name. Accessing the z/VSE host can produce an *IOException* or a *ConnectorException* error. Therefore this call must be contained in a “try-catch” clause.

```
public int modMap()
{
   ...
   /* Get name from textfield ... */
   String name = tfName.getText().toUpperCase();
   try {
      /* Change map on host */
      map.rename(name);
      ...
   }
   catch (Exception e)
   {
      ...
   }
   ...
}
```

Figure 109. Sample Applet Code for Modifying a Map

This dialog window allows you to change a map's properties:

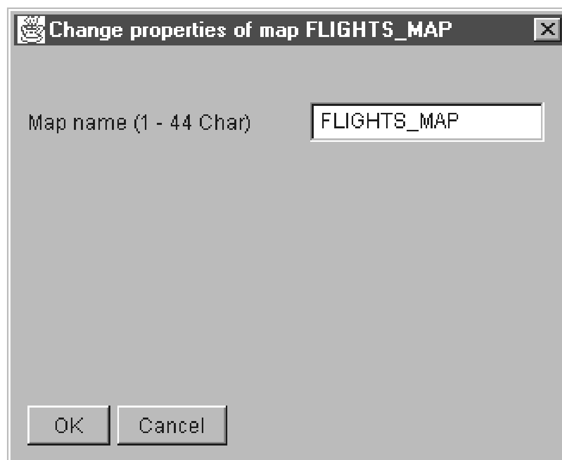


Figure 110. Window for Changing a Map's Properties

Using the Data-Mapping Applet to Modify a Map's Data Fields

The example below shows how to modify a field contained within a map, in these steps:

1. The map is retrieved by searching for the selected item in the list of maps.
2. Map-related methods (such as *setFieldName()*, *setFieldType()*, and so on) are used to change the properties of a field.
3. After modifying the field on the host, the list of local map fields is updated with the new name. In addition, the local field object is updated in the vector of map fields.

```
public int modMapfield()
{
   int i,type;
   ...
   /* Get new name from dialogbox */
   String name = tfName.getText().toUpperCase();

   /* Check if this name is already there. It's possible */
   /* to leave the name unchanged, but it's not possible */
   /* to change the name to another existing name. */
   ...

   /* Get other values from dialogbox */
   if (cb1.getState() == true)
      type = VSEVsamMap.TYPE_STRING;
   else if (cb2.getState() == true)
      type = VSEVsamMap.TYPE_BINARY;
   else if (cb3.getState() == true)
      type = VSEVsamMap.TYPE_PACKED;
   else if (cb4.getState() == true)
      type = VSEVsamMap.TYPE_SIGNED;
   else
      type = VSEVsamMap.TYPE_UNSIGNED;
   String len = tfLength.getText().toUpperCase();
   String offset = tfOffset.getText().toUpperCase();

   ...
   /* Get related map */
   i = mapList.getSelectedIndex();
   map = (VSEVsamMap)(vMaps.elementAt(i));    1
   ...
   /* Modify this field on host */
   try {
     map.setFieldName(oldName, name);
     map.setFieldType(map.getIndex(name), type);
     map.setFieldLength(map.getIndex(name),new Integer(len).intValue());
     map.setFieldOffset(map.getIndex(name),new Integer(offset).intValue());
   }
   catch (Exception e)
   {
     ...
   }

   /* Modify local field */
   i = mapFieldList.getSelectedIndex();
   mapFieldList.replaceItem(name, i);
   field = (VSEVsamField)(vMapFields.elementAt(i));
   field.setName(name);
   field.setType(type);
   field.setLength(new Integer(len).intValue());
   field.setOffset(new Integer(offset).intValue());
   ...
}
```

Figure 111. Sample Applet Code for Modifying a Map's Data Fields

1 Here, we must keep local map, view, and field instances in a vector, because it is not possible to store such objects in an AWT list.

This dialog window corresponds to the code shown above, and allows you to modify a map's data fields:

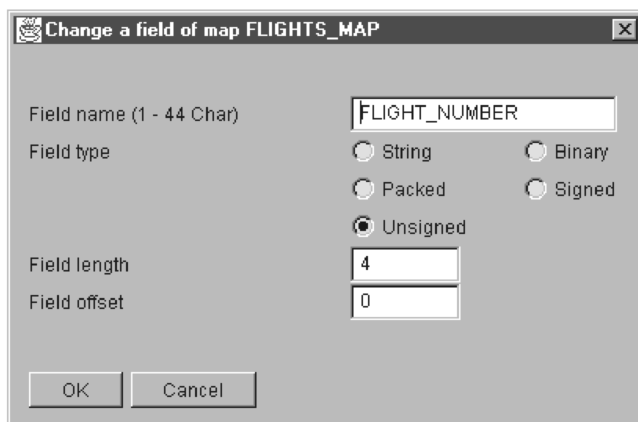


Figure 112. Window for Changing a Map's Data Fields

Note: Checking for valid length and offset values are not performed. You can also overwrite fields (specify the same offset but different lengths for different fields).

Running the Data-Mapping Applet Locally Using the AppletViewer

For a quick test of your installation, you can run the data-mapping applet *locally*: you are not required to first place the code in the z/VSE host. The VSE Connector Client online documentation contains a file **DataMapping.html**, which you can use to run the sample applet using your local appletviewer. The appletviewer is supplied with your Java installation.

```
<html>
<body>
<h2>VSAM Data Mapping Applet</h2>
<applet code="com.ibm.vse.samples.VsamMappingApplet"
        codebase="." archive="../VSEConnector.jar"
        width=460 height=460>
</applet>
</body>
</html>
```

Please note that:

- The codebase and archive tags are different from those of the z/VSE host-based HTML file. Here, we specify using the codebase tag, that all applet-related code is located in the current directory (including sub-directories).
- The archive tag points to the original **VSEConnector.jar** file that contains the VSE Java Beans class library.
- You are not required to create a second JAR file containing the applet-specific code, since the applet viewer can take these classes directly from your local file system.

Go to the **vsecon\samples** directory and call the applet in this way:

```
set classpath=.;..\VSEConnector.jar;%classpath%
AppletViewer  MappingApplet.html
```

(which assumes the **VSEConnector.jar** file is stored in the next upper directory).

The related Unix shell script would therefore be as follows:

```
#! /bin/sh
export CLASSPATH=.:../VSEConnector.jar:$CLASSPATH
appletviewer MappingApplet.html
```

Running the Sample VSAM Applet

The information in this topic is based upon a sample applet, the *VSAM applet*, that is provided with the VSE Connector Client.

Other examples of applets are similarly provided with the VSE Connector Client, such as the:

- *VsamSpaceUsage* applet (which displays the used and free VSAM space).
- *Data-Mapping Applet* (described on page "Description of the Data-Mapping Applet" on page 192).
- *DL/I Applet* (described on page "Running the Sample DL/I Applet" on page 214).

Description of the VSAM Applet

The sample VSAM applet is an example of how you can use an applet together with the DB2-Based Connector. It allows you to display and modify VSAM data stored in a sample VSAM data cluster.

When you start the VSAM applet, it connects to the DB2 Connect database-alias **db2vsewm**. The VSAM applet can then communicate with the z/VSE host database **sqlds**, *via* ***db2vsewm***.

The VSAM applet calls a sample DB2 Stored Procedure (VSAMSEL in this example) to access VSAM data via the VSAMSQL Call Level Interface (CLI). For a detailed description of the CLI, see "Using DB2 Stored Procedures to Access VSAM Data" on page 395.

Before you can run the VSAM applet, you must have customized:

- The DB2-Based Connector (see Steps 1 to 6 of Chapter 10, "Customizing the DB2-Based Connector," on page 83 for details).
- The sample DB2 Stored Procedures (see "Step 7: Customize the DB2-Based Connector for VSAM Data Access" on page 95 for details).
- DB2 Connect on your physical/logical middle-tier (see "Step 10: Install DB2 Connect and Establish Client-Host Connection" on page 96 for details). During this step, you define **sqlds** to DB2 Connect on your physical/logical middle-tier (where **sqlds** is the sample database on the z/VSE host that is used with the VSAM applet).

Note: The VSAM applet *might not run* with certain combinations of operating system and Web browser.

How the sample VSAM applet is used within a 3-tier environment, is shown in Figure 113 on page 202:

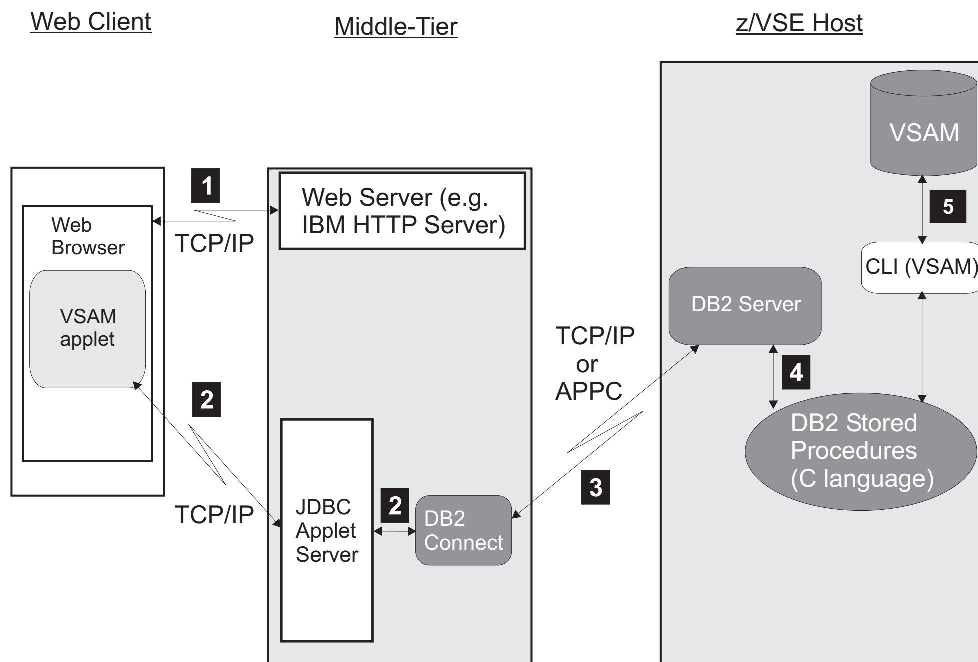


Figure 113. Using the Sample VSAM Applet to Access VSAM Data

1 The clients's Web browser requests an HTML page from the IBM HTTP Server (or another Web server) running on the physical/logical middle-tier. The HTML page contains the applet tag for the *VSAM applet*. The VSAM applet is loaded into the Java Virtual Machine of the Web browser, and starts to run.

2 The VSAM applet opens a connection to DB2 Connect running on the physical/logical middle-tier of the 3-tier platform. It does so *via the JDBC Applet Server* (where “JDBC” is an abbreviation for *Java Database Connectivity*). The use of the JDBC Applet Server overcomes the restrictions that applets can only:

- Open a new network connection to the platform from which they are downloaded (in this case, the physical/logical middle-tier).
- Access the file system of the platform from which they are downloaded (in this case, the physical/logical middle-tier).

The VSAM applet then calls a sample DB2 Stored Procedure (VSAMSEL in this example).

3 DB2 Connect communicates with the DB2 Server for VSE, via the DB2 Connect database-alias **db2vsewm**. DB2 Connect can now access to the z/VSE host database **sqlds**. It uses the DRDA (Distributed Relational Database Architecture™). The underlying protocol can be either APPC or TCP/IP.

4 The DB2 Server for VSE executes a sample DB2 Stored Procedure (VSAMSEL in this example), using the Stored Procedure Server.

5 The sample DB2 Stored Procedure can now carry out the VSAM applet's request, by accessing the VSAM data stored on the z/VSE host via VSAMSQL CLI.

Notes:

1. The WebSphere Application Server is not required on the physical/logical middle-tier of the process described above.
2. The VSE Connector Server is not required on the z/VSE host of the process described above.

Getting Started With the Sample VSAM Applet

Before you can run the VSAM applet, you must perform the steps described in this topic.

1. Create an HTML File to Call the VSAM applet

You must write an HTML file, from which the VSAM applet can be called. Each time this HTML page is displayed, the applet will be loaded and executed in the Web browser's JVM (Java Virtual Machine). The VSAM applet then connects to the physical/logical middle-tier database using JDBC.

Here is an example of such an HTML file:

```
<html>
<head>
<title>JDBC Example Applet to call a DB2 Stored Procedure</title>
</head>
<body>
<h2>JDBC Example Applet that access VSE/VSAM using the DB2-based connector</h2>
This applet lets you browse, insert, update and delete any records from the
VSE/VSAM sample cluster for the DB2-based connector. You can also browse all VSAM records
using the view OFFER. The JDBC Applet calls the corresponding Stored Procedures defined
within your DB2 Server for VSE.
<p>
<center>
<applet code="DB2ConnectorJDBCApplet.class" archive="db2applt.jar, db2java.zip"
           width=440 height=420>
</applet>
</center>
</body>
</html>
```

2. Compile VSAMSEL.C

Because DB2 Stored Procedures that use the VSAMSQL CLI are written in *C language*, you compile VSAMSEL.C (the sample DB2 Stored Procedure used in this example) using the IBM LE/VSE *C for VSE* compiler. All DB Stored Procedures must also be LE/VSE-compliant. Use job stream SKCPSTP, which is located in ICCF library 59.

The sample DB2 Stored Procedures do not contain SQL statements, since they use instead the VSAMSQL Call Level Interface (CLI). Therefore, you are not required to run the SQL precompiler. However, if you decide to include both SQL statements and VSAMSQL CLI to access your own data, you must run an additional SQL precompile step.

Here is the JCL (taken from SKCPSTP) that you can use to compile the *VSAMSEL.C* sample DB2 Stored Procedure. You compile the other sample DB2 Stored Procedures (VSAMINS.C, VSAMUPD.C, and VSAMDEL.C) in a similar way.

```
// JOB SKCPSTP COMPILE SAMPLE STORED PROCEDURE
// LIBDEF *,SEARCH=(PRD2.SCEEBASE,PRD2.DBASE,PRD1.BASE)
// SETPARM CATALOG=1
// IF CATALOG = 1 THEN
// GOTO CAT
// OPTION ERRS,SXREF,SYM,LIST,NODECK
// GOTO ENDCAT
/. CAT
// LIBDEF PHASE,CATALOG=LIB.SUBLIB
```

```
// OPTION ERRS,SXREF,SYM,NODECK,CATAL
   PHASE VSAMSEL,*
/. ENDCAT
 INCLUDE @@TRT
 INCLUDE IESVSQLO
// EXEC EDCCOMP,SIZE=EDCCOMP,PARM='LONGNAME RENT SS SOURCE            X
               INFILE(DD:PRD1.BASE(VCLIUTIL.C))'
// EXEC EDCCOMP,SIZE=EDCCOMP,PARM='LONGNAME RENT SS SOURCE            X
               INFILE(DD:PRD1.BASE(VSAMSEL.C))'
/*
// IF CATALOG NE 1 OR $MRC GT 4 THEN
// GOTO NOLNK
// EXEC EDCPRLK,SIZE=EDCPRLK,PARM='NATLANG(ENU)/UPCASE'
/*
// EXEC LNKEDT,SIZE=256K
/. NOLNK
/&
```

3. Define VSAMSEL to the DB2 Server for VSE

Now you must make VSAMSEL (the DB2 Stored Procedure used in this example) known to your DB2 Server for VSE system. To do so, you must:

1. Place the VSAMSEL *phase* (compiled in the previous step) into a library that is contained in the *Stored Procedure Server's* search path.
2. Use the CREATE PROCEDURE statement to define VSAMSEL to the database manager. You can use the job stream SKCRESTP located in ICCF library 59 for this purpose.

   Here is the CREATE PROCEDURE statement for VSAMSEL:

   ```
   CREATE PROCEDURE VSAMSEL (IN char(180),INOUT INT,OUT INT,OUT CHAR(20),-
   OUT CHAR(20),OUT CHAR(20),OUT INT,OUT INT,OUT CHAR(20),OUT CHAR(20),-
   OUT CHAR(20),OUT INT,OUT INT,OUT CHAR(20),OUT CHAR(20),OUT CHAR(20),-
   OUT INT,OUT INT,OUT CHAR(6),OUT INT,OUT CHAR(252)) EXTERNAL,LANGUAGE C,-
   STAY RESIDENT YES,SERVER GROUP,PARAMETER STYLE GENERAL
   ```

4. Define the VSAM Data Cluster

You use job SKVSSAMP (located in ICCF library 59) to:

- Create the VSAM data cluster used by the sample VSAM applet.
- Load sample records into the VSAM data cluster.
- Create a sample map and view for the VSAM data cluster, using the RECMAP command. For detailed information about RECMAP, see "Defining a Map Using RECMAP" on page 105.

```
// JOB SKVSSAMP LOAD VSE/VSAM CONNECTOR SAMPLE DATA CLUSTER
* ***************************************************************
*                                                               *
*   NOTE: IF YOU SPECIFY A DIFFERENT CATALOG THAN               *
*         VSESP.USER.CATALOG, YOU HAVE TO CHANGE THE PATH FOR   *
*         THE RECORD MAP (CATALOG/CLUSTER/MAP/VIEW) IN THE      *
*         CLIENT PROGRAMS THAT CALL THE STORED PROCEDURE AS WELL *
*         (CVSAMSEL.SQC ETC.)                                   *
*                                                               *
* ***************************************************************
*
*   DEFINING THE VSAM CONNECTOR SAMPLE DATA CLUSTER 'VCSAMPD'
*
* ***************************************************************
// EXEC IDCAMS,SIZE=AUTO
 DELETE VSAM.CONN.SAMPLE.DATA PURGE CATALOG(VSESP.USER.CATALOG)
 DEFINE CLUSTER ( -
        NAME ( VSAM.CONN.SAMPLE.DATA ) -
        RECORDS ( 30 30 ) -
        SHAREOPTIONS ( 2 ) -
        RECORDSIZE ( 120  120 ) -
        VOLUMES ( DOSRES SYSWK1 ) -
```

```
        NOREUSE -
        INDEXED -
        FREESPACE ( 15 7 ) -
        KEYS ( 4   0 ) -
        NOCOMPRESSED ) -
        DATA ( NAME ( VSAM.CONN.SAMPLE.DATA.@D@ ) -
        CONTROLINTERVALSIZE ( 4096  ) ) -
        INDEX ( NAME ( VSAM.CONN.SAMPLE.DATA.@I@ )) -
        CATALOG (VSESP.USER.CATALOG )
   IF LASTCC NE 0 THEN CANCEL JOB
/*
// OPTION STDLABEL=DELETE
                VCSAMPD
/*
// OPTION STDLABEL=ADD
// DLBL VCSAMPD,'VSAM.CONN.SAMPLE.DATA',,VSAM,                         X
               CAT=VSESPUC
/*
* ******************************************************************
*                                                                  *
*               NOW LOADING THE SAMPLE DATA                        *
*                                                                  *
* ******************************************************************
// LIBDEF *,SEARCH=(PRD1.BASE)
// EXEC VSAMSMPD,SIZE=AUTO
// ON $RC>0 GOTO FINISH
/*
* ******************************************************************
*                                                                  *
*    DEFINE MAPS AND VIEWS USING THE RECORD MAPPING UTILITY        *
*                                                                  *
* ******************************************************************
// EXEC IDCAMS,SIZE=AUTO
 RECMAP DEFINE ( MAP( USEDCARS )  -
   MAPCOLUMN( -
    ( ARTICLENO    FIELD( O(0)   L(4)   T(SINTEG))  POS(1) )   -
    ( MANUFACTURER FIELD( O(4),  L(20)  T(STRING))  POS(2) )   -
    ( TYPE         FIELD( O(24)  L(20)  T(STRING))  POS(3) )   -
    ( MODEL        FIELD( O(44)  L(20)  T(STRING))  POS(4) )   -
    ( HP           FIELD( O(64)  L(2)   T(SINTEG))  POS(5) )   -
    ( DISPLACEMENT FIELD( O(66)  L(2)   T(SINTEG))  POS(6) )   -
    ( CYLINDERS    FIELD( O(68)  L(2)   T(SINTEG))  POS(7) )   -
    ( COLOUR       FIELD( O(70)  L(20)  T(STRING))  POS(8) )   -
    ( FEATURES     FIELD( O(90)  L(20)  T(STRING))  POS(9) )   -
    ( PRICE        FIELD( O(110) L(4)   T(SINTEG))  POS(10))   -
   ) -
 ) -
 CATALOG( VSESP.USER.CATALOG )  -
 CLUSTER( VSAM.CONN.SAMPLE.DATA )
 RECMAP DEFINE ( MAP( USEDCARS )  -
   VIEW( OFFER ) -
    VIEWCOLUMN( ( ARTICLENO     REFCOLUMN( ARTICLENO    )) -
     ( MANUFACTURER  REFCOLUMN( MANUFACTURER )) -
     ( TYPE          REFCOLUMN( TYPE         )) -
     ( MODEL         REFCOLUMN( MODEL        )) -
     ( PRICE         REFCOLUMN( PRICE        )) -
   ) -
 ) -
 CATALOG( VSESP.USER.CATALOG )  -
 CLUSTER( VSAM.CONN.SAMPLE.DATA )
 RECMAP LIST (CLUSTERS)
/*
/. FINISH
/*
/&
```

5. Create the JAR File for the VSAM applet

Before running the VSAM applet, you must:

1. Create a JAR file by copying the applet-related class files into this JAR (Java Archive) file. To do so, go to the samples directory of your VSE Connector Client installation, and execute these statements:
```
call jar c0fv db2applt.jar com\ibm\vse\db2\DB2ConnectorJDBCApplet.class
   com\ibm\vse\db2\MessageDialog.class
   com\ibm\vse\db2\PowerGridLayout.class
   com\ibm\vse\db2\PowerGridLayoutInfo.class
```
2. Copy the JAR file of (1.) above, to the HTML directory of your Web Server (for example the IBM HTTP server, or Apache server) on the physical/logical middle-tier platform.

   Note: Because the VSAM applet runs in a *3-tier environment*, as an alternative to (1.) above, you could copy the class files directly to the HTML directory of your Web Server. This is because the applet code is not stored on the z/VSE host, and you are therefore not required to store class files there in a short-name archive.
3. Start the JDBC applet server on the physical/logical middle-tier platform. This server handles the requests that are initiated by the VSAM applet. You must also choose an unused TCP/IP port number that can be used by the JDBC applet server (all ports are defined in the *services* file). Therefore if you choose TCP/IP port **6789** (the default), you would enter:
```
db2jstrt 6789
```

Calling the VSAM Applet

When the HTML page is loaded, the VSAM applet is:

1. Downloaded from the physical/logical middle-tier server to a local workstation.
2. Executed in the Java Virtual Machine of a Web browser installed on the local workstation.

The VSAM applet can be called in two ways:

- Using the applet viewer directly (file **AppletViewer.exe** for Windows and OS/2), which is part of your local Java installation. To do so, from a command prompt you simply enter:
```
AppletViewer db2index.html
```
  where **db2index.html** contains the HTML tags shown in "1. Create an HTML File to Call the VSAM applet" on page 203.
- Using a Web browser. To do so, you must enter the symbolic name or IP address of your physical/logical middle-tier platform, followed by the name of the sample HTML file in your Web browser's address field. In our example, you would enter:
```
http://ebusvse/db2/db2index.html
```

After being called, the VSAM applet displays the main window shown in Figure 114 on page 207.

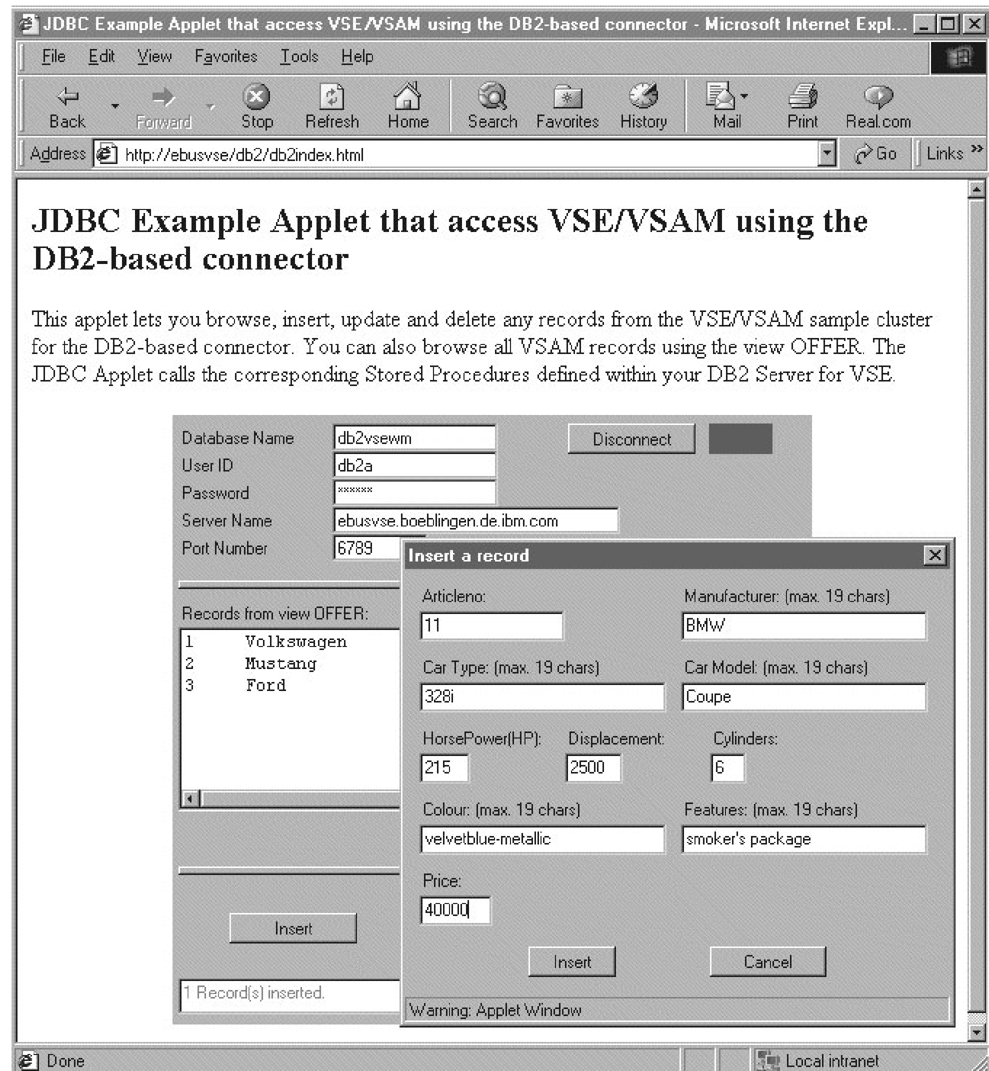


Figure 114. Window Displayed by the Sample VSAM Applet

Figure 114 includes a rectangle positioned to the right of the **Disconnect** button. This indicates the status of the connection between the VSAM applet and the **sqlds** sample database (which is accessed using the DB2 Connect database-alias **db2vsewm**). When this rectangle is:

- *green* the applet is connected to **sqlds**, and the button displays **Disconnect**.
- *red* the applet is not connected to **sqlds**, and the button displays **Connect**.

The main window shown in Figure 114 includes an *Insert a record* dialog window, which you use to insert a VSAM record in the sample VSAM cluster. You can use other similar dialog windows to refresh, update, or delete, records in the sample VSAM data cluster. Depending upon the dialog window that is currently displayed:

If You Press
: **Then ...**

Connect
: The connection to z/VSE is reestablished via DB2 Connect, and then DB2 Stored Procedure **VSAMSEL** is used together with the **OFFER** view, to select all records from a sample VSAM data cluster.

Insert The DB2 Stored Procedure **VSAMINS** is used to insert a new record into the sample VSAM data cluster.

Update
The DB2 Stored Procedure **VSAMUPD** is used to replace a record in the sample VSAM data cluster.

Delete The DB2 Stored Procedure **VSAMDEL** is used to delete a VSAM record from the sample VSAM data cluster.

Refresh
The DB2 Stored Procedure **VSAMSEL** is used, together with the **OFFER** view, to select all records from a sample VSAM data cluster.

At the bottom of Figure 114 on page 207 is a “status line”, which displays error or status messages related to the action you are currently performing.

The sample VSAM data cluster is described in “4. Define the VSAM Data Cluster” on page 204.

Description of DB2ConnectorJDBCApplet.java (the Client-Side Program)

This topic describes the *main steps* of *DB2ConnectorJDBCApplet.java*, which is used for most of the VSAM applet's functions. It runs in the Web browser's JVM of Figure 113 on page 202.

In addition, the VSAM applet uses these *helper classes*:

- *MessageDialog.java*, which displays a message-dialog window containing a specific row of text, together with an **OK** button.
- *PowerGridLayout.java* and *PowerGridLayoutInfo.java* which are a type of Java layout manager, and are much used by the VSE Connector Client samples. This Java layout manager is supplied in the *com.ibm.vse.utilities* package. You can use this Java layout manager when writing your own applications.

Step 1. Import the JDBC (Java Database Connectivity) Classes

In the first step, the JDBC classes are imported. JDBC is required for calling VSAMSEL (one of the sample DB2 Stored Procedures) that is used with the DB2 Server for VSE.

In addition, the applet-specific classes are imported. Since an applet generally *extends* the Java Applet class, the VSAM applet implements the:

- *ActionListener*, that listens to mouse-clicks and push-button actions.
- *WindowListener*, that handles window actions.

```
 ...
/* import JDBC classes */
import java.sql.*;

/* Import AWT classes */
import java.awt.*;
import java.awt.event.*;

/* Import applet classes */
import java.applet.*;
```

```
public class DB2ConnectorJDBCApplet extends Applet
implements ActionListener, WindowListener
{
```

Step 2. Load the Required JDBC Driver Class

In the second step, the required JDBC driver class is loaded. This is normally done in a static section. The *net* driver class is used, which ensures that the applet can run on any client workstation.

```
...
// register the JDBC driver with DriverManager
static
{
   try
   {
      Class.forName ("COM.ibm.db2.jdbc.net.DB2Driver");
   }
   catch (Exception e)
   {
      System.out.println ("\n  Error loading DB2 Driver...\n");
      e.printStackTrace ();
   }
} // end static block
...
```

Step 3. Implement the init() Method

In the third step, the *init()* method is implemented. The *init()* method is called from the Web browser when the VSAM applet is first started. A frame is created, that is the parent of the various dialogs used for refreshing, inserting, updating, or deleting, VSAM records in the sample VSAM data cluster (described in "4. Define the VSAM Data Cluster" on page 204).

```
...
 public void init()
 {
    /* Create a frame that is needed for the dialogs */
    /* to insert/update/delete records and to display message dialogs */
    f = new Frame();

    msgDialog = new MessageDialog(f, true);
    pgl = new PowerGridLayout(100, 66);
    setLayout(pgl);

    displayMainDialog();
    repaint();
 }
 ...
```

Step 4. Establish the Connection to z/VSE Database via DB2 Connect

In the fourth step, a connection is established to the z/VSE host database **sqlds**, via the DB2 Connect database-alias **db2vsewm**. DB2 Connect can then route database requests to the z/VSE host.

The following JDBC URL format is used to set up the connection:

```
<protocol>:<subprotocol>://<hostname or tcpip address>:<port number>/<database name>
```

For the database specified under *<database name>*, the JDBC *getConnection()* call supplies:

- the JDBC URL
- a user ID

- a password

The *getConnection()* call retrieves these values from the text fields of the Main Window's *connect* section.

```
...

public void connectToDB()
{
   String url    = "jdbc:db2://" + DBServerAddr.getText() + ":"
                     + DBServerPort.getText() + "/"
                     + DBName.getText();
   ...
   try
   {
      // connect with user-provided username and password
      con = DriverManager.getConnection(url, userid.getText(), passw.getText());
   }
   catch (SQLException sqlExc)
   {
      ...
   }
   ...
}
```

Step 5. Call VSAMSEL

In the final step, VSAMSEL (the sample DB2 Stored Procedure used in this example) is called.

VSAMSEL is also called when the end-user presses the **Refresh** button.

```
...
/**
 * call DB2-based connector Stored Procedure on z/VSE
 * This will retrieve all records from view OFFER, which then will be
 * put in the corresponding listbox of the applet.
 * The Stored Procedure VSAMSEL uses the VSAMSQL CLI to access the VSAM
 * records.
 */
 public void callStpVSAMSEL()
 {
    ...
 }
```

For details of how JDBC issues a call to a DB2 Stored Procedure, refer to the JDBC application samples supplied with the VSE Connector Client online documentation. (The online documentation is described in "Using the Online Documentation Options" on page 26).

Description of VSAMSEL

This topic describes the *main steps* of VSAMSEL (the sample DB2 Stored Procedure used in this example) that runs on the z/VSE host of Figure 113 on page 202. The other sample DB2 Stored Procedures (VSAMINS, VSAMUPD, and VSAMDEL) are not described in this topic.

VSAMSEL is called when either the **Connect** or **Refresh** buttons of Figure 114 on page 207 are pressed, and it demonstrates how VSAM data is accessed via the VSAMSQL CLI (Call Level Interface).

Step 1. Include Header File iesvsql.h in VSAMSEL

In the first step, the header file **iesvsql.h** is included in VSAMSEL. This header file is required by *all* DB2 Stored Procedures that use the VSAMSQL CLI interface to access VSAM data clusters.

Header file **iesvsql.h**:

- Maintains all function prototypes and VSAMSQL CLI definitions.
- Contains the prototype for the *check_error()* function, which provides examples of error-handling routines for VSAMSQL CLI calls. The *check_error()* function is called after each VSAMSQL CLI call.

```
...
// include VSAMSQL CLI
#include "iesvsql.h"
// include error utility functions
#include "vcliutlh.h"
...
```

Step 2. Initialize the VSAMSQL CLI Environment

In the second step, the *VSAMSQL CLI* (Call Level Interface) environment is initialized. The VSAMSQL CLI is described in “Using DB2 Stored Procedures to Access VSAM Data” on page 395.

These *handles* are initialized:

- The *environment handle*, which provides access to global information (such as valid connection handles and active connection handles).
- The *connection handle*, which provides access to connection information (such as the valid statement and descriptor handles on the connection).
- The *statement handle*, which provides access to statement information (such as error messages and status information for VSAMSQL statement processing).

```
....

/***************************************************************/
/*      initialize VSAMSQL CLI Environment                     */
/***************************************************************/
// allocate Environment
rc = VSAMSQLAllocHandle(VSAMSQL_HANDLE_ENV,VSAMSQL_NULL_HANDLE, &hEnv);
cont = check_error(VSAMSQL_HANDLE_ENV,hEnv,rc,
                   o_sqlstate,o_message,&o_native_error,SYSLST,"VSAMSEL-1");

//allocate Connection
if (cont == STP_CONT) {
   rc = VSAMSQLAllocHandle(VSAMSQL_HANDLE_DBC,hEnv,&hDBC);
   cont = check_error(VSAMSQL_HANDLE_DBC,hDBC,rc,
                      o_sqlstate,o_message,&o_native_error,SYSLST,VSAMSEL-2");
} // end if

//allocate Statement
if (cont == STP_CONT) {
   rc = VSAMSQLAllocHandle(VSAMSQL_HANDLE_STMT,hDBC,&hStmt);
   cont = check_error(VSAMSQL_HANDLE_STMT,hStmt,rc,
                      o_sqlstate,o_message,&o_native_error,SYSLST,"VSAMSEL-3");
} // end if
...
```

Step 3. Initiate the Read of the VSAM Records

In the third step, the query statement to initiate the read of the VSAM records is prepared and executed. All fields contained in the view *i_recordmap* are selected (the view *OFFER* is passed from the client program within this example).

During the third step:

1. The *VSAMSQLPrepare()* function call associates the SELECT statement string with the statement handle.
2. *VSAMSQLBindParameter()* binds *i_lastkey* to the WHERE clause within the SELECT statement (where **?** is the corresponding placeholder).
3. *i_lastkey* is passed as an input parameter for VSAMSEL (the sample DB2 Stored Procedure).
4. *VSAMSQLExecute()* executes the statement, after it was successfully prepared. The statement handle *hStmt* qualifies the query statement.

```
...

/**************************************************************/
/*      prepare and execute query statement                   */
/**************************************************************/
if (cont == STP_CONT) {
   sprintf(vsamsqlstmt, "SELECT * FROM %s WHERE ARTICLENO > ? ",
           i_recordmap);
   rc = VSAMSQLPrepare(hStmt, vsamsqlstmt, VSAMSQL_NTS);
   cont = check_error(VSAMSQL_HANDLE_STMT,hStmt,rc,
                      o_sqlstate,o_message,&o_native_error,SYSLST,"VSAMSEL-4");
} // end if

// Bind local Variables to Stmt
if (cont == STP_CONT) {
   rc = VSAMSQLBindParameter(hStmt,1,VSAMSQL_PARAM_INPUT,
                             VSAMSQL_C_LONG,VSAMSQL_INTEGER,
                             0,0,&i_lastkey,0,NULL);
   cont = check_error(VSAMSQL_HANDLE_STMT,hStmt,rc,
                      o_sqlstate,o_message,&o_native_error,SYSL"VSAMSEL-5");

} // end if

if (cont == STP_CONT) {
   rc = VSAMSQLExecute(hStmt);
   cont = check_error(VSAMSQL_HANDLE_STMT,hStmt,rc,
                      o_sqlstate,o_message,&o_native_error,SYSLST,"VSAMSEL-6");
} // end if
...
```

Step 4. Obtain the Results of the Query Statement

In the fourth step, since the query statement has executed, the results can now be fetched. But primarily all columns from the result set must be associated with a local variable (*tmp.articleno*). This is done using *VSAMSQLBindCol()* function calls, which must be issued for each column in the result set.

Within the *for()-loop*, a single record from the view *OFFER* is retrieved using *VSAMSQLFetch()*.

VSAMSEL can fetch and return three result records to the client program:

1. If less than three result records exist, condition VSAMSQL_NO_DATA_FOUND appears.
2. The *VSAMSQLFetch()* function call outside the *for()-loop* checks if more than three result records exist.

3. If more than three result records exist, *o_resultrows* is increased by one. This value is used as an indicator by the client program to determine if VSAMSEL should be called, in order to retrieve the next records.

```
...

/****************************************************************/
/*      retrieve result set                                     */
/****************************************************************/
// bind columns to local variables and retrieve results
if (cont == STP_CONT) {
   // bind parameter articleno
   rc = VSAMSQLBindCol(hStmt,1,VSAMSQL_C_LONG,
                       &tmp.articleno,0,&buf_len);
   cont = check_error(VSAMSQL_HANDLE_STMT,hStmt,rc,
                      o_sqlstate,o_message,&o_native_error,SYSLST,"VSAMSEL-7");
} // end if
...

// fetch result row(s)
for (i=0; i < NUM_ROWS; i++)
{
   if (cont == STP_CONT) {
      rc = VSAMSQLFetch(hStmt);
      cont = check_error(VSAMSQL_HANDLE_STMT,hStmt,rc,
                     o_sqlstate,o_message,&o_native_error,SYSLST,"VSAMSEL-C");
   } // end if

   // if successful - store retrieved columns
   if ( cont == STP_CONT && rc != VSAMSQL_NO_DATA_FOUND) {
      o_resultrows ++;
      o_parm[i].articleno = tmp.articleno;
      strcpy(o_parm[i].manufacturer,tmp.manufacturer);
      strcpy(o_parm[i].type,tmp.type);
      strcpy(o_parm[i].model,tmp.model);
      o_parm[i].price     = tmp.price;
   } // end if
   ...
} // end for

// check if more result exist
if (cont == STP_CONT) {
   rc = VSAMSQLFetch(hStmt);
   cont = check_error(VSAMSQL_HANDLE_STMT,hStmt,rc,
                o_sqlstate,o_message,&o_native_error,SYSLST,"VSAMSEL-D");
   if (rc != VSAMSQL_NO_DATA_FOUND)
      o_resultrows ++;
} // end if
...
```

Step 5. Deallocate the VSAMSQL CLI Environment

In the fifth step, since all VSAMSQL CLI processing has been completed, the VSAMSQL CLI environment can be de-allocated using the function *VSAMSQLFreeHandle()*. This is done in the reverse order to the allocation step ("Step 2. Initialize the VSAMSQL CLI Environment" on page 211).

```
                              ...
/****************************************************************/
/*      free VSAMSQL CLI Environment                            */
/****************************************************************/
// free handle Statement
rc = VSAMSQLFreeHandle(VSAMSQL_HANDLE_STMT,hStmt);
cont = check_error(VSAMSQL_HANDLE_STMT,hStmt,rc,
                o_sqlstate,o_message,&o_native_error,SYSLST,"VSAMSEL-E");

// free handle Connection
rc = VSAMSQLFreeHandle(VSAMSQL_HANDLE_DBC,hDBC);
```

```
cont = check_error(VSAMSQL_HANDLE_DBC,hDBC,rc,
                   o_sqlstate,o_message,&o_native_error,SYSLST,"VSAMSEL-F");

// free handle Environment
rc = VSAMSQLFreeHandle(VSAMSQL_HANDLE_ENV,hEnv);
cont = check_error(VSAMSQL_HANDLE_ENV,hEnv,rc,
                   o_sqlstate,o_message,&o_native_error,SYSLST,"VSAMSEL-10");
...
```

Step 6. Assign Local Output Variables to Host Output Variables

In the final step, the local output variables that are to be returned to the client program, must be assigned to the corresponding host output variables.

```
...
/***************************************************************/
/*  assign number rows with valid data, if >=4 more data exists */
/***************************************************************/
*(VSAMSQLINTEGER *)argv[2] = o_resultrows;       // rows returned

// copy result rows to output parameters
j=0;
for (i=0; i < NUM_ROWS; i++)
{
   *(VSAMSQLINTEGER *)argv[j+3] = o_parm[i].articleno;
   strcpy(argv[j+4], o_parm[i].manufacturer);
   strcpy(argv[j+5], o_parm[i].type);
   strcpy(argv[j+6], o_parm[i].model);
   *(VSAMSQLINTEGER *)argv[j+7] = o_parm[i].price;
   j = j + 5;
} // end for
...
```

Running the Sample DL/I Applet

The information in this topic is based upon a sample applet, the *DL/I applet*, that is provided with the VSE Connector Client.

Other examples of applets are similarly provided with the VSE Connector Client, such as the:

- *VsamSpaceUsage* applet (which displays the used and free VSAM space).
- *Data-Mapping Applet* (described on page “Description of the Data-Mapping Applet” on page 192).
- *VSAM Applet* (described on page “Running the Sample VSAM Applet” on page 201).

Description of the DL/I Applet

The sample DL/I applet is an example of how you can use an applet together with the DB2-Based Connector, and allows you to display and modify DL/I data stored in a sample DL/I database.

When you start the DL/I applet, it connects to the DB2 Connect database-alias **db2vsewm**. The DL/I applet can then communicate with the z/VSE host database **sqlds** via **db2vsewm**.

The DL/I applet then calls a sample DB2 Stored Procedure (in this example DLIREAD) to access DL/I data via the *AIBTDLI interface*. For a description of the AIBTDLI interface, “Overview of the AIBTDLI Interface” on page 403.

Before you can run the DL/I applet, you must have customized:

- The DB2-Based Connector (see Steps 1 to 6 of Chapter 10, "Customizing the DB2-Based Connector," on page 83 for details).
- The sample DB2 Stored Procedures (see "Step 7: Customize the DB2-Based Connector for VSAM Data Access" on page 95 for details).
- DB2 Connect on your physical/logical middle-tier (see "Step 10: Install DB2 Connect and Establish Client-Host Connection" on page 96 for details). During this step, you define **sqlds** to DB2 Connect on your physical/logical middle-tier (where **sqlds** is the sample database on the z/VSE host that is used with the VSAM applet).

Note: The DL/I Applet *might not run* with certain combinations of operating system and Web browser.

How the sample DL/I applet is used within a 3-tier environment, is shown in Figure 115:

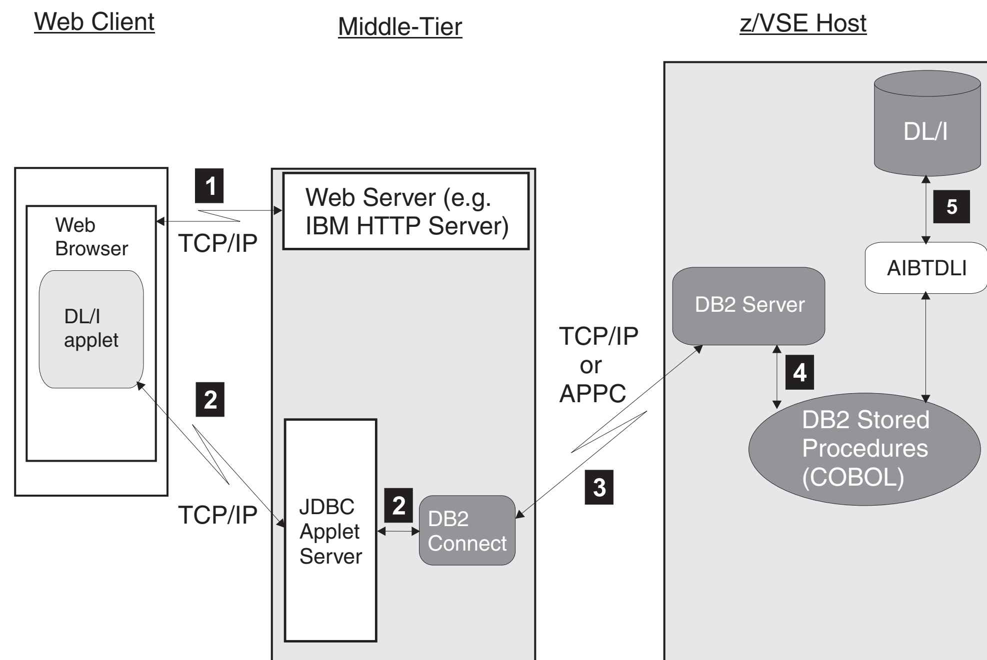


Figure 115. Using the Sample DL/I Applet to Access DL/I Data

1 The clients's Web browser requests an HTML page from the IBM HTTP Server (or another Web server) running on the physical/logical middle-tier. The HTML page contains the applet tag for the *DL/I applet*. The DL/I applet is loaded into the Java Virtual Machine of the Web browser, and starts to run.

2 The DL/I applet opens a connection to DB2 Connect running on the physical/logical middle-tier. It does so *via the JDBC Applet Server* (where "JDBC" is an abbreviation for *Java Database Connectivity*). The use of the JDBC Applet Server overcomes the restrictions that applets can only:

- Open a new network connection to the platform from which they are downloaded (in this case, the physical/logical middle-tier).
- Access the file system of the platform from which they are downloaded (in this case, the physical/logical middle-tier).

The DL/I applet then calls a sample DB2 Stored Procedure (DLIREAD in this example).

3 DB2 Connect communicates with the DB2 Server for VSE, via the DB2 Connect database-alias **db2vsewm**. DB2 Connect can now access the z/VSE host database **sqlds**. It uses the DRDA (Distributed Relational Database Architecture). The underlying protocol can be either APPC or TCP/IP.

4 DB2 Server for VSE manages the execution of a sample DB2 Stored Procedure (DLIREAD in this example), using the Stored Procedure Server.

5 The DB2 Stored Procedure can now execute the DL/I applet's request by accessing the DL/I data stored on the z/VSE host via the *AIBTDLI interface* (described in "Overview of the AIBTDLI Interface" on page 403).

Notes:

1. The WebSphere Application Server is not required on the physical/logical middle-tier of the process described above.
2. The VSE Connector Server is not required on the z/VSE host of the process described above.

Getting Started With the Sample DL/I Applet

Before you can run the DL/I applet, you must perform the steps described in this topic.

1. Create an HTML File to Call the DL/I applet

You must write an HTML file, from which the DL/I applet can be called. Each time this HTML page is displayed, the applet will be loaded and executed in the Web browser's JVM (Java Virtual Machine). The applet then connects to the physical/logical middle-tier database using JDBC.

Here is an example of such an HTML file:

```
<html>
<head>
<title>
JDBC Example Applet that accesses DL/I using the DB2-based connector
</title>
</head>
<body>
<h2>JDBC Example Applet that accesses DL/I data</h2>
This applet lets you browse, insert, update and delete segments from/into the
DL/I inventory sample database via the DB2-based connector.
The JDBC Applet calls the corresponding Stored Procedures defined
within your DB2 Server for VSE using JDBC interface.
The Stored Procedures use the AIBTDLI interface to access the DLI/VSE data.
<p>
<center>
<applet code="DB2DLIConnectorJDBCApplet.class"
           archive="db2dliapplt.jar, db2java.zip"
           width=440 height=420>
</applet>
</center>
</body>
</html>
```

2. Compile DLIREAD.C

Because DB2 Stored Procedures that uses the AIBTDLI interface are written in *COBOL*, you compile DLIREAD.C (the DB2 Stored Procedure used in this example) using the IBM LE/VSE *COBOL* compiler. The compile job stream, SKDLICMP, is located in ICCF library 59.

DLIREAD.C does not contain SQL statements, since it uses instead the AIBTDLI interface. Therefore, you are not required to run the SQL precompiler. However, if you decide to include both SQL statements and DL/I calls to access your own data, you must run an additional SQL precompile step.

To compile the other sample DB2 Stored Procedures (DLIUPINS.C and DLIDEL.C) that are used for accessing DL/I data, use the same skeleton SKDLICMP.

3. Define DLIREAD to the DB2 Server for VSE

Now you must make DLIREAD known to your DB2 Server for VSE system. To do so, use the job skeleton SKDLISTP (located in VSE/ICCF library 59) to create DB2 Stored Procedures used for accessing the DL/I sample database.

1. Place the DLIREAD *phase* (compiled in the previous step) into a library that is contained in the *Stored Procedure Server's* search path.
2. Use the CREATE PROCEDURE statement to define DLIREAD to the database manager. You can use the job stream SKDLISTP located in ICCF library 59 for this purpose.

   Here is an example of the CREATE PROCEDURE statement for DLIREAD:

   ```
   CREATE PROCEDURE DLIREAD (INOUT CHAR(6),OUT SMALLINT,
                             OUT CHAR(6),OUT CHAR(25),
                             OUT INT,OUT CHAR(6),OUT CHAR(6),
                             OUT CHAR(6),OUT CHAR(25),
                             OUT INT,OUT CHAR(6),OUT CHAR(6),
                             OUT CHAR(6),OUT CHAR(25),
                             OUT INT,OUT CHAR(6),OUT CHAR(6),
                             OUT CHAR(4),OUT CHAR(120))
                             EXTERNAL,LANGUAGE COBOL,
                             STAY RESIDENT YES,
                             SERVER GROUP,
                             PARAMETER STYLE GENERAL;
   ```

4. Define the DL/I Database

Before you can use DL/I applet, you must have defined a DL/I database. You use job SKDLISMP (located in ICCF library 59) to define and load the *sample* DL/I database.

5. Create the JAR File for the DL/I applet

Before running the DL/I applet, you must:

1. Create a JAR file by copying the applet-related class files into this JAR (Java Archive) file. To do so, go to the samples directory of your VSE Connector Client installation, and execute these statements:

   ```
   call jar c0fv db2dliapplt.jar com\ibm\vse\db2\DB2DLIConnectorJDBCApplet.class
      com\ibm\vse\db2\MessageDialog.class
      com\ibm\vse\db2\PowerGridLayout.class
      com\ibm\vse\db2\PowerGridLayoutInfo.class
   ```

2. Copy the JAR file of (1.) above to the HTML directory of your Web Server (for example the IBM HTTP server, or Apache server) on the physical/logical middle-tier platform.

   Note: Because the DL/I applet runs in a *3-tier environment*, as an alternative to (1.) above, you could copy the class files directly to the HTML directory of

your Web Server. This is because the applet code is not stored on the z/VSE host, and you are therefore not required to store class files there in a short-name archive.

3. Start the JDBC applet server on the physical/logical middle-tier platform. This server handles the requests that are initiated by the DL/I applet. You must also choose an unused TCP/IP port number that can be used by the JDBC applet server. Therefore if you choose TCP/IP port **6789** (the default), you would enter:

```
db2jstrt 6789
```

Calling the DL/I Applet

When the HTML page is loaded, the DL/I applet is:

1. Downloaded from the physical/logical middle-tier server to a local workstation.
2. Executed in the Java Virtual Machine of a Web browser installed on the local workstation.

The DL/I applet can be called in two ways:

- Using the applet viewer directly (file **AppletViewer.exe** for Windows and OS/2), which is part of your local Java installation. To do so, from a command prompt you simply enter:

  ```
  AppletViewer index.html
  ```

  where **index.html** contains the HTML tags shown in “1. Create an HTML File to Call the DL/I applet” on page 216.
- Using a Web browser. To do so, you must enter the symbolic name or IP address of your physical/logical middle-tier platform, followed by the name of the sample HTML file in your Web browser's address field. In our example, you would enter:

  ```
  http://ebusvse/db2dli/index.html
  ```

After being called, the DL/I applet displays the main window shown in Figure 116 on page 219.

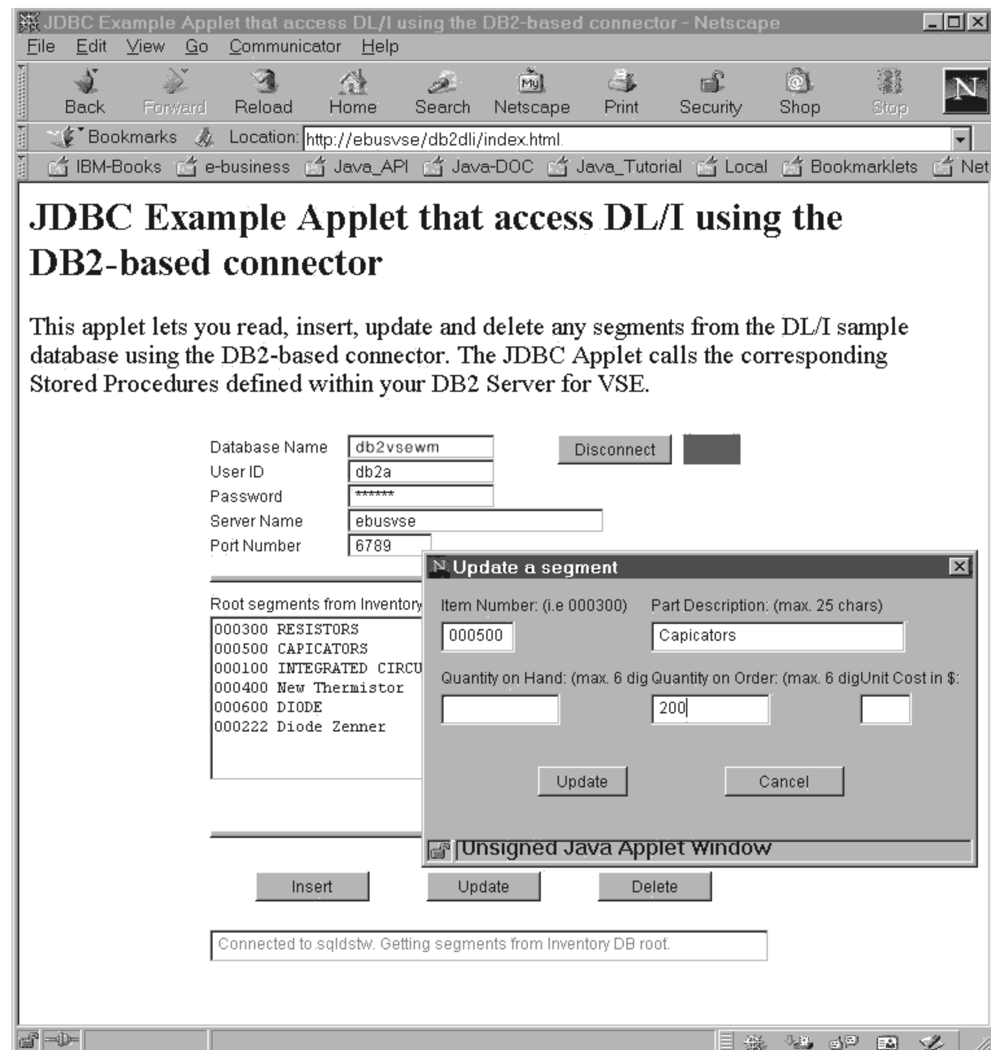


Figure 116. Window Displayed by the Sample DL/I Applet

Figure 116 includes a rectangle positioned to the right of the **Disconnect** button. This rectangle indicates the status of the connection between the DL/I applet and the **sqlds** sample database on the z/VSE host (which is accessed using the DB2 Connect database-alias **db2vsewm**). When this rectangle is:

- *green* the applet is connected to **sqlds**, and the button displays **Disconnect**.
- *red* the applet is not connected to **sqlds**, and the button displays **Connect**.

The main window shown in Figure 116 includes an *Update a segment* dialog window, which you use to insert a DL/I segment in the sample DL/I database. You can use other similar dialog windows to refresh, insert, or delete, segments in the sample DL/I database. Depending upon the dialog window that is currently displayed:

Press Then ...

Connect
: The connection to z/VSE is re-established via DB2 Connect, and then the DB2 Stored Procedure **DLIREAD** is used to read all segments in the sample DL/I database.

Insert The DB2 Stored Procedure **DLIUPINS** is used to insert a new segment in the sample DL/I database.

Update
The DB2 Stored Procedure **DLIUPINS** is used to update a segment in the sample DL/I database.

Delete The DB2 Stored Procedure **DLIDEL** is used to delete a segment in the sample DL/I database.

Refresh
The DB2 Stored Procedure **DLIREAD** is used to read all segments in the sample DL/I database.

At the bottom of Figure 116 on page 219 is a “status line”, which displays error or status messages related to the action you are currently performing.

Description of DB2DLIConnectorJDBCApplet.java (the Client-Side Program)

This topic describes the *main steps* of *DB2DLIConnectorJDBCApplet.java*, which is used for most of the DL/I applet's functions. It runs in the Web browser's JVM of Figure 115 on page 215.

In addition, method *callStpDLIREAD()* is the part of the DL/I applet that performs the Read/Browse of a DL/I database. All input and output variables must be set to the appropriate parameter placeholders (which are shown in the DL/I applet samples as question marks).

Step 1. Prepare the SQL Statement to Call DLIREAD

Before the first step can be carried out, a connection object *con* must exist to the z/VSE host. This connection object is opened using method *connectToDB()* (described in “Step 4. Establish the Connection to z/VSE Database via DB2 Connect” on page 209).

```
public void callStpDLIREAD()
{
   CallableStatement stmt; // SQL Statement Handle
   String sql =            // JDBC Stored Procedure Call String
       "Call " + name + "(?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)";

   // Prepare Stored Procedure Call Statement
   try
   {
      stmt = con.prepareCall(sql);
      // add header lines into applet list box
      RecordList.add("ITEMNO ITEM-DESCRIPTION          QUAN-H QUAN-O PRICE");
      RecordList.add("===================================================");
      do
      {
         //** set input variables **
         stmt.setString (1, nextkey);

         //** register output parameters **
         // next key returned from procedure
         stmt.registerOutParameter(1, Types.CHAR);
         stmt.registerOutParameter(2, Types.SMALLINT);

         // 3 data rows that can hold data from root segment of
         // inventory DB
        for (int i=3; i <= 13; i=i+5)
```

```
{
   stmt.registerOutParameter ( i  , Types.CHAR);
   stmt.registerOutParameter ( i+1, Types.CHAR);
   stmt.registerOutParameter ( i+2, Types.INTEGER);
   stmt.registerOutParameter ( i+3, Types.CHAR);
   stmt.registerOutParameter ( i+4, Types.CHAR);
}

// Variables for error handling
stmt.registerOutParameter(18, Types.CHAR); // return code
stmt.registerOutParameter(19, Types.CHAR); // error message
```

Step 2. Call DLIREAD

In the second step, DLIREAD (the sample DB2 Stored Procedure used in this example) is called by executing the statement created in Step 1.

```
stmt.executeQuery();
```

Step 3. Check the Return Code from DLIREAD

In the third step, the return code is checked that was returned by DLIREAD.

```
// Get return code from Stored Procedure
ret_code = stmt.getString(18);

// Check if the Stored Procedure returned an error
if ((ret_code.compareTo("0000") == 0) ||
    (ret_code.substring(0,1).compareTo("G") == 0))
{
   // no error
   // get number of result rows (1-3)
   res_rows  = stmt.getShort(2);
```

Providing an error is not found, the output variables from DLIREAD are retrieved. The output parameters are read, and the results then displayed by the DL/I applet.

```
// Get returned fields from root segment
for (int i = 0, j = 3; i < 3;  i++, j=j+5)
{
   itemno[i]       = stmt.getString(j);
   itemdesc[i]     = stmt.getString(j+1);
   unit_cost[i]    = stmt.getInt(j+2);
   quan_hand[i]    = stmt.getString(j+3);
   quan_ord[i]     = stmt.getString(j+4);

   // add fields from root segment into listbox
   if (i < res_rows)
   {
      String temp = Integer.toString(unit_cost[i]);
      while(temp.length()<5)
         temp += ' ';
      RecordList.add(itemno[i] + " " + itemdesc[i]
          + " " + quan_hand[i] + " "
          + quan_ord[i] + " " + temp + "$ ");
   }
}
```

The program now:

1. Checks to see if there are further results. If there *are* further results, a DLIREAD call is made.
2. Uses the *nextkey* variable to check if there are further data result rows. If there *are* further data rows, DLIREAD is called again for the next 3 rows.

```
nextkey    = stmt.getString(1);
if (nextkey == "000000")
   moreresults = false;
else  // more data available in the database
```

```
      moreresults = true;
   }
   else
   {
      // error occured
      // check if DB is empty
     if ((ret_code.substring(0,2) == "GB") &&
         (RecordList.getItemCount() == 0))
        setStatus("Inventory Database is empty!!");
     else {
        DLIerrmsg = stmt.getString(19);  // get DLI error message
        setStatus("AIBTDLI Return Code: 0x" + ret_code);
        msgDialog.setTitle("Stored Procedure Error");
        System.out.println("DLI Error: " + DLIerrmsg);
        msgDialog.setMessage("Check Java Console for DLZ messages.");
        msgDialog.setVisible(true);
     } // end if
   } // end if
 }
 while ( moreresults && ret_code.compareTo("0000") == 0);
```

Step 4. Reset the Connection to the sqlds Database

In the final step, whenever the end-user presses the **Disconnect** button of Figure 116 on page 219, the program resets the connection to the **sqlds** database located on the z/VSE host. **sqlds** is accessed via the DB2 Connect database-alias **db2vsewm**.

```
        if ((ret_code.compareTo("0000") == 0) ||
            (ret_code.substring(0,2).compareTo("GB") == 0))   // no error
        {
           setStatus(RecordList.getItemCount()-2 +
                    " segments retrieved from Inventory DB.");
        }

        // close Statement Handle
        stmt.close();
      }
      catch (SQLException sqlExc)   // handle SQL exceptions
      {
        closeConnection();
        ...
        return;
      }
    }
   ...
 }
 ...
```

For details of how to insert, update, or delete DL/I segments, you should refer to the complete DL/I applet source code, supplied with the VSE Connector Client online documentation. (The online documentation is described in “Using the Online Documentation Options” on page 26).

Description of DLIREAD

This topic describes the *main steps* of DLIREAD, which is the sample DB2 Stored Procedure used in this example. It is written in COBOL and issues DL/I calls via the *AIBTDLI interface* (described in “Overview of the AIBTDLI Interface” on page 403). DLIREAD runs on the z/VSE host of Figure 115 on page 215, and is called when the **Connect** or **Refresh** buttons of Figure 116 on page 219 are pressed.

The complete source code is contained in file DLIREAD.COB, which is supplied with the VSE Connector Client. The other sample DB2 Stored Procedures (DLIUPINS and DLIDEL) are not described in this topic.

Using one call to DLIREAD, you can read and return up to three segments from the sample DL/I database:

- The data is returned via DLIREAD's output parameters, where each data field of a DL/I segment has a corresponding output parameter.
- Other output parameters include *error-indicator variables*, that contain information about errors that might have occurred in DB2 or DL/I, or indicate the status of a request from a DB2 Stored Procedure.
- The DL/I applet contains a variable which it uses to check if there are further (more than three) DL/I segments waiting to be retrieved. If more than three records *are* to be retrieved, the client program calls DLIREAD again, using an ascending key.

Step 1. Define Variables for AIBTDLI, and I/O Area

In the first step, these are defined:

- The variables for the DL/I AIBTDLI interface.
- The I/O area used to communicate with DL/I.

```
...
*
*  UNQUALIFIED SSA FOR GETTING FIRST ROOT SEGMENT
*
   77  SSAUNQ        PIC X(9)  VALUE 'STPIITM  '.
*
*  QUALIFIED SSA FOR GETTING ROOT SEGMENT VIA KEY
*
   01  SSAQUAL.
     02  FILLER    PIC X(19) VALUE 'STPIITM (STQIINO = '.
     02  ROOTKEY   PIC X(6).
     02  FILLER    PIC X(1)  VALUE ')'.
*
* I-O AREA FOR RECEIVING ALL SEGMENTS
*
   01  IOAREA        PIC X(160).
   01  STPIITM         REDEFINES IOAREA.
     02  ITNUMB    PIC X(6).
     02  ITDESC    PIC X(25).
     02  IQOH      PIC X(6).
     02  IQOR      PIC X(6).
     02  FILLER    PIC X(6).
     02  IUNIT     PIC 9(6).
     02  FILLER    PIC X(105).
  LINKAGE SECTION.
```

Step 2. Define the Parameters for DLIREAD

In the second step, the parameters for DLIREAD are defined. These parameters build the interface between the DL/I applet, and the DB2 Server for VSE running on the z/VSE host.

In the example provided here, the parameters for three rows are defined. Therefore, up to three rows can be retrieved from the DL/I sample database, using a single call to DLIREAD.

```
*
* STORED PROCEDURE PARAMETERS
*
  01  NEXT-KEY     PIC X(6).
  01  NUM-ROWS     PIC S9(4)   COMP.
```

```
* FIRST RESULT ROW
  01  ITEM-NUMB1   PIC X(6).
  01  ITEM-DESC1   PIC X(25).
  01  UNIT-COST1   PIC S9(6)   COMP.
  01  QUAN-HAND1   PIC X(6).
  01  QUAN-ORD1    PIC X(6).

* SECOND RESULT ROW
  01  ITEM-NUMB2   PIC X(6).
  01  ITEM-DESC2   PIC X(25).
  01  UNIT-COST2   PIC S9(6)   COMP.
  01  QUAN-HAND2   PIC X(6).
  01  QUAN-ORD2    PIC X(6).

* THIRD RESULT ROW
  01  ITEM-NUMB3   PIC X(6).
  01  ITEM-DESC3   PIC X(25).
  01  UNIT-COST3   PIC S9(6)   COMP.
  01  QUAN-HAND3   PIC X(6).
  01  QUAN-ORD3    PIC X(6).
  01  RET-CODE     PIC X(4).
  01  DLI-ERR-MSG  PIC X(120).
  COPY DLIAIB.
  ...
```

Step 3. Define DLIREAD's Parameters to COBOL

In the third step, in the PROCEDURE DIVISION of DLIREAD, the parameters are defined that represent the external interface to DLIREAD.

```
 ....
*****************************
* BEGIN OF PROGRAMMING SECTION
*****************************
  PROCEDURE DIVISION USING
            NEXT-KEY NUM-ROWS
            ITEM-NUMB1 ITEM-DESC1
            UNIT-COST1 QUAN-HAND1
            QUAN-ORD1
            ITEM-NUMB2 ITEM-DESC2
            UNIT-COST2 QUAN-HAND2
            QUAN-ORD2
            ITEM-NUMB3 ITEM-DESC3
            UNIT-COST3 QUAN-HAND3
            QUAN-ORD3
            RET-CODE DLI-ERR-MSG.
```

Step 4. Obtain the Results of the Query Statement

In the fourth step, the DL/I processing starts:

1. DLIREAD issues up to three DL/I calls, to retrieve the segments.
2. If there are no further segment to retrieve, after the third call has been made, the processing ends .
3. If there are result-rows to be retrieved, DLIREAD is called another three times.

```
* *****************************************************
* SCHEDULE THE PSB
* *****************************************************
   SCHEDULE-PSB.
   MOVE 'STBICLG' TO PSB-NAME.
   CALL 'AIBTDLI' USING FUNCT-PCB, PSB-NAME, ADDRESS OF DLIAIB.
   IF AIBFCTR NOT = LOW-VALUES  GO TO BASERR
   SET ADDRESS OF PCB-PTRS TO AIBPCBAL.
   SET ADDRESS OF INV-L-PCB TO B-PCB1-PTRS.

* *****************************************************
```

```
 * DO A GET NEXT CALL TO RETRIEVE THE FIRST ROOT SEGMENT
 * ******************************************************
    MOVE NEXT-KEY TO ROOTKEY.
    IF NEXT-KEY = '000000'
       CALL 'AIBTDLI' USING FUNCT-GN, INV-L-PCB, IOAREA, SSAUNQ
```

The calls to DL/I are made for a maximum of 3 segments. The values are returned to the DL/I applet via the corresponding parameters defined for the DLIREAD.

```
 ...
 * ******************************************************
 * DO A GET UNIQUE CALL TO RETRIEVE A SEGMENT VIA KEY
 * ******************************************************
    ELSE
      CALL 'AIBTDLI' USING FUNCT-GU, INV-L-PCB, IOAREA, SSAQUAL
    END-IF.
    IF AIBFCTR NOT = LOW-VALUES
      GO TO BASERR.

 * *****************************************************
 * ASSIGN VALUES TO OUTPUT PARAMETERS FOR ROW 1
 * *****************************************************
    ADD 1 TO COUNTR.
    MOVE ITNUMB  TO ITEM-NUMB1.
    MOVE ITDESC  TO ITEM-DESC1.
    MOVE IQOH    TO QUAN-ORD1.
    MOVE IQOR    TO QUAN-HAND1.
    MOVE IUNIT   TO UNIT-COST1.
    MOVE COUNTR TO NUM-ROWS.
```

```
Issue second call to the stored procedure to retrieve second result row:
```

```
 * ****************************************************
 * ISSUE GET NEXT CALL
 * ****************************************************
  CALL 'AIBTDLI' USING FUNCT-GN, INV-L-PCB, IOAREA,
          SSAUNQ.
  IF AIBFCTR NOT = LOW-VALUES
    GO TO BASERR.

 * *****************************************************
 * ASSIGN VALUES TO OUTPUT PARAMETERS FOR ROW 2
 * *****************************************************
   ADD 1 TO COUNTR.
   MOVE ITNUMB  TO ITEM-NUMB2.
   MOVE ITDESC  TO ITEM-DESC2.
   MOVE IQOH    TO QUAN-ORD2.
   MOVE IQOR    TO QUAN-HAND2.
   MOVE IUNIT   TO UNIT-COST2.
   MOVE COUNTR TO NUM-ROWS.
```

Below, the third (and final) call to DLIREAD is issued. The third result-row is retrieved.

```
 * *****************************************************
 * ISSUE GET NEXT CALL
 * *****************************************************
   CALL 'AIBTDLI' USING FUNCT-GN, INV-L-PCB, IOAREA, SSAUNQ.
   IF AIBFCTR NOT = LOW-VALUES
     GO TO BASERR.

 * *****************************************************
 * ASSIGN VALUES TO OUTPUT PARAMETERS FOR ROW 3
 * *****************************************************
   ADD 1 TO COUNTR.
   MOVE ITNUMB  TO ITEM-NUMB3.
   MOVE ITDESC  TO ITEM-DESC3.
```

```
   MOVE IQOH    TO QUAN-ORD3.
   MOVE IQOR    TO QUAN-HAND3.
   MOVE IUNIT   TO UNIT-COST3.
   MOVE COUNTR TO NUM-ROWS.
```

Step 5. Check for Further DL/I Segments

In the fifth step, a check is made to see if there are any further DL/I segments. The variable NEXTKEY is set to the next DL/I segment key that satisfies the initial condition.

```
 * ****************************************************
 *  DETERMINE IF FURTHER SEGMENTS EXIST AND SAVE KEY
 * ****************************************************
   CALL 'AIBTDLI' USING FUNCT-GN, INV-L-PCB, IOAREA, SSAUNQ.*

 * MORE SEGMENTS AVAIL - SAVE KEY FOR THE NEXT PROCEDURE CALL
 *
      MOVE INV-KEY-FBCK(1:6) TO NEXT-KEY.
      GO TO END-PROC.
```

Step 6. Run the Error-Handling Routines

In the final step, DLIREAD uses error-handling routines to determine if the request has been successful.

```
*
 *  DLI-CALL ERROR HANDING
  ...
```

Chapter 18. Using Java Servlets to Access Data

This chapter contains these main topics:

- "How Servlets Are Used in 3-Tier Environments"
- "Compiling and Calling Servlets" on page 229
- "Example of How to Implement a Servlet" on page 230

How Servlets Are Used in 3-Tier Environments

Servlets are used in the physical/logical middle-tier of a 3-tier environment shown in Figure 117 on page 228 in this way:

© Copyright IBM Corp. 2000, 2013

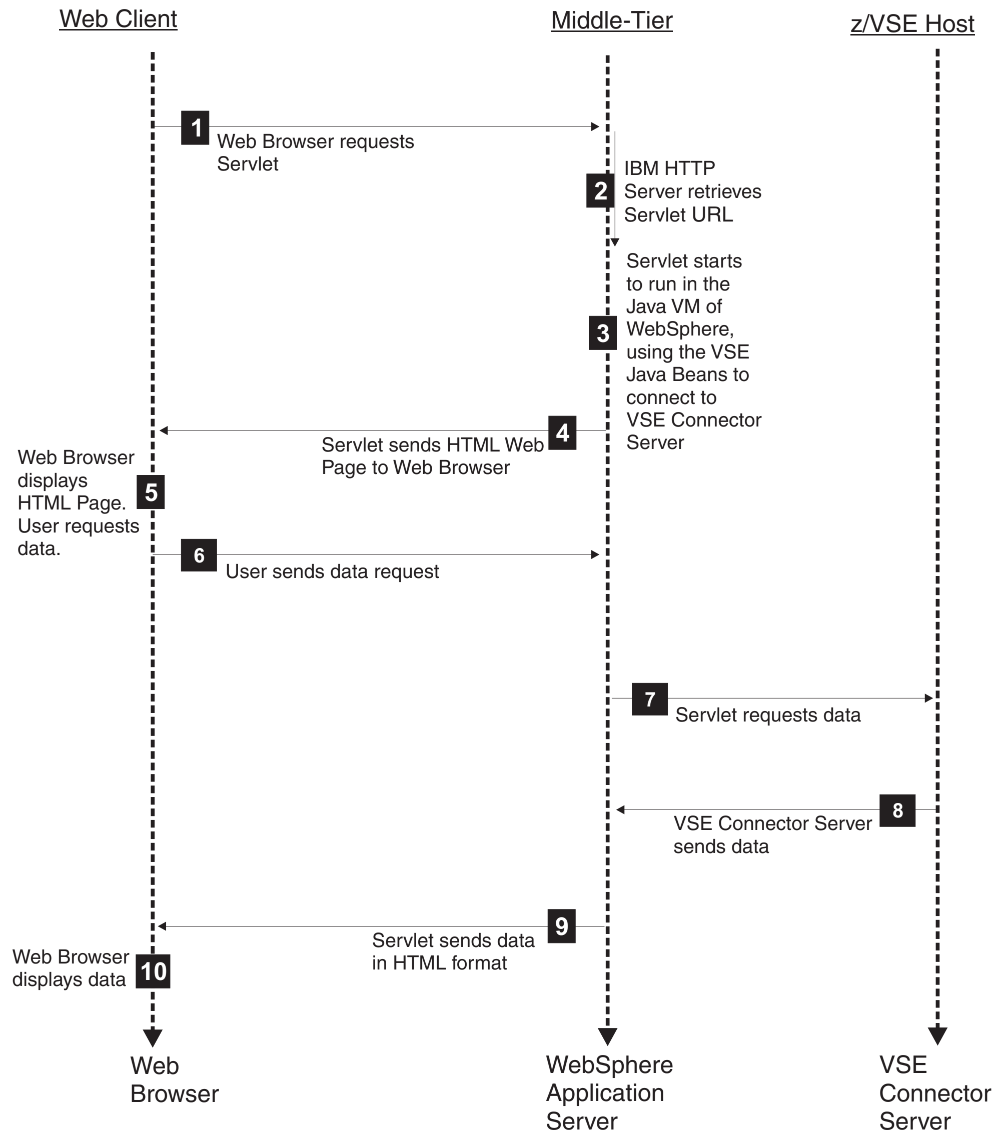


Figure 117. How Servlets Are Used in the z/VSE 3-Tier Environment

Figure 117 assumes that a VSE Connector Server running on the z/VSE host is used for obtaining data.

HTTP Sessions are used between the Web Client and the physical/logical middle-tier for sending and receiving data. Socket connections are used between the physical/logical middle-tier and the z/VSE host for sending and receiving data.

The number of each list item below describes a step shown in Figure 117:

1 The client's Web browser sends a servlet request to the IBM HTTP Server running on the physical/logical middle-tier.

2 The IBM HTTP Server retrieves the servlet URL and passes this to the WebSphere Application Server.

3 The WebSphere Application Server runs the servlet in its Java Virtual

Machine (JVM). The servlet also uses the VSE Java Beans class library (**VSEConnector.jar**) to build a connection to the VSE Connector Server running on the z/VSE host.

4 The servlet sends the dynamically-created HTML Web Page to the client's Web browser, via TCP/IP.

5 The client's Web browser displays the Web Page. The end-user now requests data that is stored on the z/VSE host.

6 The client's Web browser sends the request for data to the servlet, which is still loaded in the Java Virtual Machine of the WebSphere Application Server.

7 The servlet uses the VSE Java Beans to communicate with the VSE Connector Server running on the z/VSE host, requesting the VSE-based data.

8 The VSE Connector Server obtains the data and sends it to the servlet.

Notes:

1. The VSE Connector Server can be used for accessing VSE/VSAM, VSE/POWER, VSE/ICCF, or Librarian data.
2. An alternate method for accessing VSAM data stored on the z/VSE host is to use a DB2 Stored Procedure on the physical/logical middle-tier, which communicates with the VSAM file on the z/VSE host. This is described in "Using DB2 Stored Procedures to Access VSAM Data" on page 395.

9 The servlet generates a dynamic Web Page consisting of HTML code, and sends this together with the requested data back to the client's Web browser.

10 The client's Web browser re-displays the Web Page together with the requested data.

Compiling and Calling Servlets

To compile a servlet, you require servlet-related Java classes. These classes are normally provided together with your WebSphere Application Server installation. These classes are contained in a JAR file that must be included into your local classpath (for your compilations), as well as in the WebSphere Application Server's classpath.

Servlets can then be called in various ways. This topic describes the most commonly-used way:

From the command line

You must simply enter:

```
http://server_host_name/servlet_engine_name/servlet_name
```

For example, from the command line of your Web browser you could enter:

```
http://MyComputer/webapp/VseServletEngine/MyServlet
http://9.164.123.456/webapp/VseServletEngine/MyServlet
```

How the WebSphere Application Server Stores Session Information

These TCP/IP connections are short-lived:
- From the Web client to the physical/logical middle-tier platform.
- From the physical/logical middle-tier to the z/VSE host.

The servlet's *HTTPSession* object allows Web clients to be identified over a series of HTTP requests. WebSphere does this internally, for example by storing a "cookie" on the Web client.

Each time the servlet is invoked, it checks whether the Web client already had a connection in the past, by searching for the *HTTPSession* object (which might be bound to the “cookie”). If the session does already exist, the servlet then checks if the session contains information about a previously-used session.

The *httpSession* instance is created when the servlet is invoked the first time (in this case, the session does not already exist). Therefore two actions must be performed:
1. Create the httpSession instance and store it in the HTTPServletRequest.
2. Define certain session properties, and store them in the httpSession.

Figure 118 shows the "logical view" of how session information is stored and processed. *httpSession* contains the properties, and is stored on the Web client as a cookie.

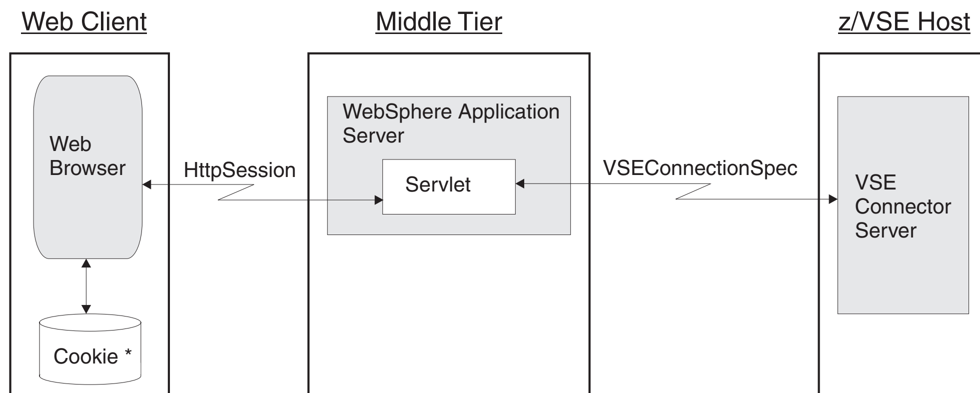


Figure 118. How Session Information Is Re-Used by the WebSphere Application Server

Example of How to Implement a Servlet

This topic describes the *FlightOrderingServlet* sample, which is supplied as part of the online documentation. This servlet sample provides the capability to maintain flights and orders information in a simple VSAM-based flight booking system. It first defines, and then uses, two VSAM clusters which hold the data.

For details of how to define the data maps and views for the clusters used in this sample, refer to “Running the Sample Data-Mapping Applet” on page 192.

General Description of the Sample Servlet

The servlet is implemented in the Java source file **FlightOrderingServlet.java**. This source file implements the main servlet code, together with some inner classes, to handle flights, orders, create the VSAM clusters, and so on.

The servlet performs this general processing:

1. A Web page is displayed in which the user can choose between several actions. From this first Web page, the user can create the required VSAM clusters and fill them with data, order a flight, and cancel a flight.
2. When the user clicks **Order a flight**, a table with all available flights is displayed. Here, the user can select a flight number. When selecting a flight number, the servlet displays another Web page containing the properties of this flight, and the window controls required for the user to be able to place an order. The user can enter:
   - Name
   - Number of seats to order
   - Whether or not the reserved seats should be in the non-smoking area.
3. After clicking the push button to order the flight, the properties of the VSAM record that contains the flight details, are updated. In addition, a new record is created in the ORDERS cluster.

The following topics describe the most important parts of the sample servlet's Java code. The complete source code is contained in the VSE Connector Client online documentation.

You might also refer to the VSE Connector Client online documentation for these additional servlet samples:

- *SearchServlet*: a servlet that shows you how to search VSE library systems (POWER, ICCF, VSE Libraries, VSAM) for files, including those containing specific text.
- *SdlServlet*: a servlet that shows you how to display a list of VSE phases loaded into the SVA.

Creating the VSAM Clusters for the Sample

These are the two VSAM clusters you require, in order to run the sample. Both are predefined in the VSAM catalog VSAM.VSESP.USER.CATALOG (VSESPUC) of VSE/ESA 2.5 onwards:

```
FLIGHT.ORDERING.FLIGHTS (FLIGHTS) - KSDS
Offset Length Type      Key    Field Name     Description
-------------------------------------------------------------
0      4      UNSIGNED  yes    FLIGHT_NUMBER  Flight Number
4      20     STRING    no     START          Start
24     20     STRING    no     DESTINATION    Destination
44     5      STRING    no     DEPARTURE      Departure (hh:mm)
49     5      STRING    no     ARRIVAL        Arrival   (hh:mm)
54     4      UNSIGNED  no     SEATS          Seats
58     4      UNSIGNED  no     RESERVED       Seats reserved
62     4      PACKED    no     PRICE          Price
66     20     STRING    no     AIRLINE        Airline
```

Figure 119. VSAM Structure of FLIGHT.ORDERING.FLIGHTS

```
Offset Length Type      Key    Field Name     Description
---------------------------------------------------------------
0      20     STRING    no     FIRST_NAME     First Name
20     20     STRING    no     LAST_NAME      Last Name
40     4      UNSIGNED  no     FLIGHT_NUMBER  Flight Number
44     4      UNSIGNED  no     SEATS          Seats
48     1      BINARY    no     NON_SMOKE      Non Smoke 0=no
```

Figure 120. VSAM Structure of FLIGHT.ORDERING.ORDERS

HTML Constructs Used With the Sample

As servlets are used for generating dynamic *Web pages*, HTML syntax is therefore an essential part of every servlet. Here are two HTML constructs that are typically used in servlets:

How a Servlet Can Create Tables in HTML

A table is defined in HTML in this general way:

```
<table>
<tr><td>"First table cell"</td><td>"Second cell"</td></tr>  (First row)
<tr>  .... </tr>                                            (Second row)
...
</table>
```

A servlet can create *dynamic tables* by writing values of program variables into a table. A dynamic table might look like this:

```
out.println("<table>");
out.println("<tr><td>" + string1 + </td></tr>");
out.println("<tr><td>" + string2 + </td></tr>");
...
out.println("</table>");
```

where:

- *string1* and *string2* are string variables.
- *out* is the PrintWriter instance that was obtained from *HttpServletResponse* in the servlet's *doGet()* or *doPost()* methods.

Using Forms to Obtain a User's Input

After showing a Web page, the servlet must be able to obtain the user's input in order to process the next servlet request. This is usually done using *forms*, which can display window controls such as text fields or push buttons. When the user performs the action associated with the form (usually a push button), the servlet is called again with this action and input parameters taken from text fields, check boxes and other input controls.

```
out.println("<form action=\"/servlet/FlightOrderingServlet\" method=get>");  1
out.println("<input type=hidden name=\"action\" value=\"order3\">");
out.println("<input type=hidden name=\"flight\" value=\"" +
                                                   flightNumber + "\">");
out.println("<table>");
out.println("<tr><td><b>First Name:</b></td>");
out.println("<td><input size=20 maxlength=20 name=\"firstname\"></td>");
out.println("</tr>");

out.println("<tr><td><b>Last Name:</b></td>");
out.println("<td><input size=20 maxlength=20 name=\"lastname\"></td>");
out.println("</tr>");
...

out.println("<tr><td><b>Non smoking:</b></td>");
out.println("<td><input type=checkbox name=\"nonsmoke\"
                                             value=\"yes\">Yes</td>");
out.println("</tr>");
out.println("</table><p>");
out.println("<input type=submit value=\"Order It!\">");
out.println("</form>");
```

Note: The string used here to perform a servlet call is dependent on the:
- Version of the WebSphere Application Server you have installed.
- Name you have defined for your servlet engine.

Figure 121. Example of a Servlet Using Forms to Obtain a User's Input

By specifying *method=get* (at **1**), the *doGet()* method of the servlet will be called. However:
- You could also specify *method=post*, which would cause the *doPost()* method to be called.
- The difference between *method=get* and *method=post* is that:
  – Using *doGet()* will display the generated servlet invocation string in the Web browser's address/location field.
  – Using *doPost()* will suppress the Web browser's address/location field.
- You are recommended to use *doPost()* if the form contains any password fields. Passwords will then appear as clear text, when using *doGet()*.

The HTML code shown in Figure 121 will display the window controls shown below:

Figure 122. Example of Using Forms to Display Window Controls

If the user presses the push-button **Order It!**, this servlet invocation string will be generated:

```
http://computername/servlet/FlightOrderingServlet?action=order3&flight=34
&firstname=name1&lastname=name2&seats=1&nonsmoke=no
```

where:

- *name1* is the string that was entered in the *firstname* field.
- *name2* is the string that was entered in the *lastname* field.
- *computername* is the name of your workstation in your network, or its IP address.

You could also enter this string manually in your Web browser's address/location field.

Sample Servlet Step 1: Display a List of Flights

In this step, a list of available flights is displayed, from which the user can place an order. The *selectRecords()* method receives all records in the FLIGHT.ORDERING.FLIGHTS cluster, and displays them.

```
public class FlightOrderingServlet extends HttpServlet
{
   // Names of the Clusters and Maps.
   String   vsamCatalog    = "VSESP.USER.CATALOG";
   String   flightsCluster = "FLIGHT.ORDERING.FLIGHTS";
   String   ordersCluster  = "FLIGHT.ORDERING.ORDERS";
   String   flightsMapName = "FLIGHTS_MAP";
   String   ordersMapName  = "ORDERS_MAP";
   ...

   public void doOrderStep1(PrintWriter out,VSESystem system)
   {
      VSEVsamCluster flights = null;
      VSEVsamMap  flightsMap = null;
      FlightsListener fl;

      // create the instances of the flights cluster with its map
      flights = new VSEVsamCluster(system,vsamCatalog, flightsCluster);   1
      flightsMap = flights.getVSEVsamMap(flightsMapName);
      ...

      try
      {
         // use the Listener to build the table
         fl = new FlightsListener(out);   2
         flights.addVSEResourceListener(fl);

         // select all records of the flights cluster (no filter)
         flights.selectRecords(flightsMap);   3
         flights.removeVSEResourceListener(fl);
      }
      catch(Throwable t)
      {
         ...
      }
      ...
   }
   ...
```

Figure 123. Sample Servlet Code for Displaying a List of Flights

The numbers below refer to the numbers in Figure 123:

1 A local instance *flights* of a *VSEVsamCluster* is created. Next, a new map instance *flightsMap* with the specified name is created. This map will be

filled with the actual data fields, during Step 2 below. During Step 1, however, these objects are only local, and no host access has been made.

2 A *VSEResourceListener* is created, to receive the VSAM records from the host.

3 All records are retrieved from the host. When the *selectRecords()* method returns, all VSAM records are displayed in the current HTML page.

Sample Servlet Step 2: Get Flight Instances from the Host

The following code is an extract of the inner class *FlightsListener*. The *listAdded()* method is a callback function that is called for each received VSAM record instance. For further details about callback functions, see "Using the Callback Mechanism of VSE Java Beans" on page 146.

```
public void listAdded(VSEResourceEvent event)
{
   String flightNumber, start, destination, departure;
   String arrival, price, airline;

   // The event data has to be a VSEVSamRecord
   if (!(event.getData() instanceof VSEVsamRecord))
     return;

   // Get the record -> it is a record of the flights cluster
   VSEVsamRecord flight = (VSEVsamRecord)(event.getData());   1
   try
   {
      // Get the fields of the record ...
      flightNumber = ((Integer)flight.getKeyField(0)).toString();
      start        = fli ght.getField(1).toString().trim();
      destination  = flight.getField(2).toString().trim();
      departure    = flight.getField(3).toString().trim();
      arrival      = flight.getField(4).toString().trim();
      price        = ((Integer)flight.getField(7)).toString().trim();
      airline      = flight.getField(8).toString().trim();

      // Write out an HTML line in the table
      ...  2

   }
   catch(Throwable t)
   {
         ...
  }
```

Figure 124. Sample Servlet Code for Getting Flight Instances from the Host

The numbers below refer to the numbers in Figure 124:

1 An explicit cast is necessary to get the *VSEVsamRecord* instance from the *VSEResourceEvent* data.

2 Refer to the VSE Connector Client online documentation for further code details.

The Java code described in Figure 124 displays this Web page:

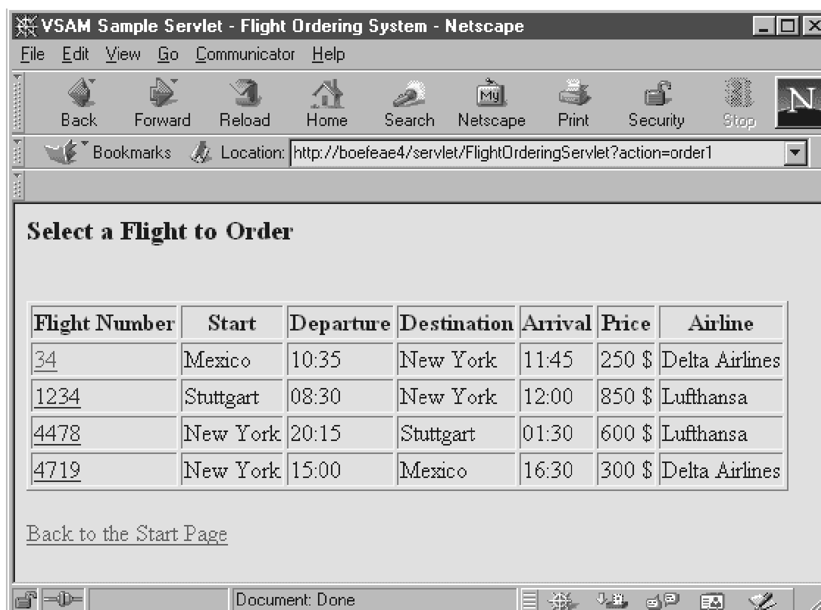


Figure 125. Flight Order Selection Window, As Generated by the Sample Servlet

In Figure 125, the first servlet call has finished. This means, the servlet code is still loaded into memory on the physical/logical middle-tier platform by the WebSphere Application Server, but no further processing will be done until the servlet is called again.

Notes:

1. The next servlet call is in fact a new program call, and there is no way to store information in global variables, from one call to the next call. Therefore, all input parameters must be passed as *servlet parameters*.
2. In fact, there is one property stored over the lifetimes of a servlet: the VSE host connection specification (*VSEConnectionSpec*). This is stored in the client connection (*HttpSession*), which is always the same for *all* servlet calls. For further details, refer to the online documentation provided with the VSE Connector Client (see “Using the Online Documentation Options” on page 26).

Sample Servlet Step 3: Display the Properties of a Flight

In the method described here, the properties of a flight are displayed, together with the controls required to place an order. An HTML form is used to:

- Get the user's input.
- Initiate a new servlet call.

```
public void doOrderStep2(PrintWriter out,VSESystem system,
                         Hashtable parameters)
{
   VSEVsamCluster flights = null;
   VSEVsamMap     flightsMap = null;
   VSEVsamRecord  flight = null;
   int flightNumber, price, seats, reserved;
   String start, destination, departure, arrival, airline;

   // create the instances of the flights cluster and its map
   flights = new VSEVsamCluster(system, vsamCatalog, flightsCluster);
   flightsMap = flights.getVSEVsamMap(flightsMapName);

   // get a instance of a record of the flights cluster
   flight = flights.getVSEVsamRecord(flightsMap);   1

   // get the parameters
   flightNumber = 0;
   try
   {
      flightNumber = Integer.parseInt(getParameterValue(parameters,   2
                                               "FLIGHT",true));
   }
   catch (Throwable t) {}
   ...

   // get the record and display the properties
   try
   {
      // set the key to identify the locat record with the record
      // on the host
      flight.setKeyField(0,new Integer(flightNumber));   3

      // now get the other fields
      start        = flight.getField(1).toString().trim();
      destination  = flight.getField(2).toString().trim();
      departure    = flight.getField(3).toString().trim();
      arrival      = flight.getField(4).toString().trim();
      seats        = ((Integer)flight.getField(5)).intValue();
      reserved     = ((Integer)flight.getField(6)).intValue();
      price        = ((Integer)flight.getField(7)).intValue();
      airline      = flight.getField(8).toString().trim();

      // display the fields
      ...

      // check if enough seats are available
      ...

      // write out a form where the user can enter its name and select
      // the number of seats to order
      out.println("<form action=\"/servlet/FlightOrderingServlet\"   4
                                                   method=get>");
      ...
      out.println("<input type=submit value=\"Order It!\">");
      out.println("</form><p>");
   }
   catch(Throwable t)
   {
      ...
   }
   ...
}
```

Figure 126. Sample Servlet Code for Displaying Properties of a Flight

The numbers below refer to the numbers in Figure 126:

1 A local instance of class *VSEVsamRecord* is created. No properties have been assigned at this point. This code simply creates an object. You could also use a constructor of the class to obtain the same result.

2 Function *getParameterValue()* is used to obtain the flight number from the form. The source code of this method is shown in the VSE Connector Client online documentation.

3 This statement now identifies one specific record in the FLIGHTS cluster. The flight number is provided in the user's input. Now, all other properties of this flight can be retrieved from this object.

4 Window controls required for the user to place the order, are displayed.

The coding described in Figure 126 on page 237 will display this Web page:

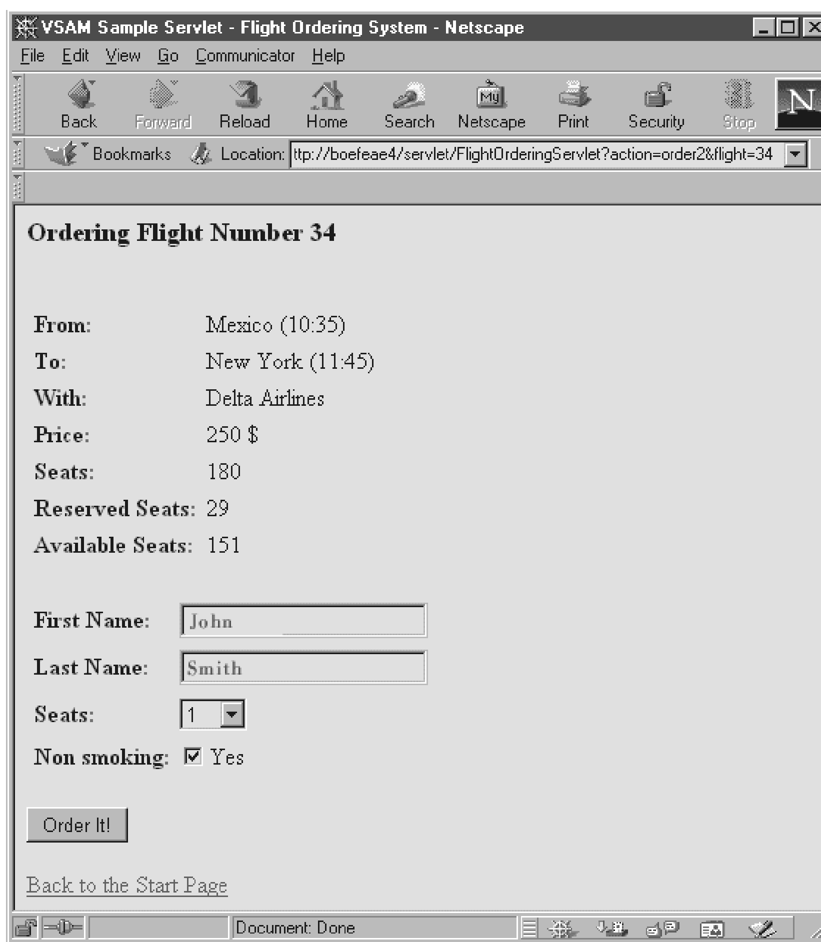


Figure 127. Flight Order Entry Window, As Generated by the Sample Servlet

Sample Servlet Step 4: Place an Order

The method described in this topic is used to place an order. It is called when the user presses the **Order It** button:

1. The user's input is read from the form.
2. The flight's properties are updated (the number of reserved seats for this flight is increased by the number of ordered seats).
3. A new record is added to the ORDERS cluster.

```
public void doOrderStep3(PrintWriter out, VSESystem system,
                         Hashtable parameters)
{
   VSEVsamCluster flights = null, orders = null;
   VSEVsamMap flightsMap = null, ordersMap = null;
   VSEVsamRecord flight = null, order = null;
   int flightNumber, seatsToOrder, seats, reserved, price;
   String firstName, lastName;
   boolean nonSmoke, ok;

   // get the parameters from the form
   try
   {
      flightNumber = Integer.parseInt(   1
                         getParameterValue(parameters,"FLIGHT",true));
      seatsToOrder = Integer.parseInt(
                         getParameterValue(parameters,"SEATS",true));
      firstName = getParameterValue(parameters,"FIRSTNAME",false);
      lastName = getParameterValue(parameters,"LASTNAME",false);

      // Check user input ...
      ...

      nonSmoke = false;
      if (getParameterValue(parameters,"NONSMOKE",true) != null)
        nonSmoke = true;
   }
   catch(Throwable t)
   {
      // if not all parameters has been specified -> redisplay Step 2
      doOrderStep2(out,system,parameters);
      return;
   }

   // create instances of the flight cluster and its map
   flights = new VSEVsamCluster(system,vsamCatalog,flightsCluster);   2
   flightsMap = flights.getVSEVsamMap(flightsMapName);

   // get a instance of a record of the flights cluster
   flight = flights.getVSEVsamRecord(flightsMap);

   // create instances of the orders cluster and its map
   orders = new VSEVsamCluster(system,vsamCatalog,ordersCluster);   3
   ordersMap = orders.getVSEVsamMap(ordersMapName);

   // get a instance of a record of the orders cluster
   order = orders.getVSEVsamRecord(ordersMap);

   // Write HTML header
   ...
  try
   {
      // get the flight record
      // set the key to identify the local record with
      // the record on the host
      flight.setKeyField(0,new Integer(flightNumber));   4
      // get some fields of interest
      seats       = ((Integer)flight.getField(5)).intValue();
      reserved    = ((Integer)flight.getField(6)).intValue();
      price       = ((Integer)flight.getField(7)).intValue();
```

```
        // check if enough seats are available
        ...

        // display the order properties
        ...
        // use the OrderCounter to get teh highest record number.
        // This is necessarry because RRDS can only add a record
        // with a non existing record number
        OrderCounter oc = new OrderCounter();   5
        // select all records to find the highest record number
        orders.addVSEResourceListener(oc);
        orders.selectRecords(ordersMap);
        orders.removeVSEResourceListener(oc);

        // now create the new order
        ok = createOrder(out,order,oc.getHighestRecNo()+1,   6
                 firstName,lastName,flightNumber,seatsToOrder,nonSmoke);

        out.println("</table><p>");

        // check if creating was ok ...
        if (ok)
        {
           // now update the flight record's fields
           // increase the reserved seats by the number of seats to order
           reserved += seatsToOrder;

           // set the field in the local record
           flight.setField(6,new Integer(reserved));

           // and commit the changes to make them permanent
           flight.commit();   7         }
     }
     catch (Throwable t)
     {
        ...
     }

     // write out some status information and HTML footer
     ...

  }
```

The numbers below refer to the numbers in the above example:

1 Obtains the servlet parameters from the *form*.

2 Creates a local instance of the FLIGHTS cluster and one record belonging to this cluster. Up to this point, no data has been associated with these objects.

3 Creates a local instance of the ORDERS cluster and one record belonging to this cluster. Up to this point, no data has been associated with these objects.

4 Sets the key field of the FLIGHTS record, which is required before any further processing of the record can take place. The information is required internally in order to access the record on the host.

5 For RRDS clusters, new records are added using unique relative record numbers. Therefore, the highest relative record number must be determined, before a new record is added. The *OrderCounter* class is used to receive all records, count them, and return the highest relative record number.

6 Method *createOrder()* is now used to add a new record to the ORDERS cluster (in the code that follows).

7 The *commit()* method is used to make the updates permanent.

Sample Servlet Step 5: Create a New Flight

The method described here adds a new record to the FLIGHTS cluster. If the record already exists (as identified by its key), an appropriate error message is generated.

```
public boolean createFlight(PrintWriter out,VSEVsamRecord flight,
                        int flightNumber,String start,String
                        destination,String departure, String arrival,
                        int seats,int reserved,int price,String airline)
throws IOException,ConnectorEx ception
{
   try
   {
       // Set the key of the record
       flight.setKeyField(0,new Integer(flightNumber));   1

       // Set all other fields
       flight.setField(1,makeString(start,20));
       flight.setField(2,makeString(destination,20));
       flight.setField(3,makeString(departure,5));
       flight.setField(4,makeString(arrival,5));
       flight.setField(5,new Integer(seats));
       flight.setField(6,new Integer(reserved));
       flight.setField(7,new Integer(price));
       flight.setField(8,makeString(airline,20));
       // try to add this record
       flight.add();   2

       // Add new row to table
       out.println("<tr><td>" + flightNumber + "</td>");
       out.println("<td>" + start + "</td>");
       out.println("<td>" + destination + "</td>");
       out.println("<td>" + departure + "</td>");
       ...
       out.println("</tr>");
   }
   catch (AlreadyExistentException e)
   {
      // already existing
      return(false);
   }
   return(true);
}
```

Figure 128. Sample Servlet Code for Creating a New Flight

The numbers below refer to the numbers in Figure 128:

1 For *KSDS clusters* all key fields must be set for a given record, before performing any actions against the record.

2 Adds the new flight record to the cluster.

Sample Servlet Step 6: Create a New Order

The method described here adds a new record to the ORDERS cluster.

```
public boolean createOrder(PrintWriter out,VSEVsamRecord order,int recNo,
                        String firstName,String lastName,int flightNumber,
                        int seats,boolean nonSmoke)
throws IOException,ConnectorException
{
   byte[] b = new byte[1];
   try
   {
      // Set the relative record number
      order.setRelRecNo(recNo);   1

      // set all other fields
      order.setField(0,makeString(firstName,20));
      order.setField(1,makeString(lastName,20));
      order.setField(2,new Integer(flightNumber));
      order.setField(3,new Integer(seats));
      if(nonSmoke)
        b[0] = (byte)0x01;
      else
        b[0] = (byte)0x00;
      order.setField(4,b);

      // Add the new record
      order.add();   2

      // Write table row
      out.println("<tr><td>"+firstName+"</td>");
      out.println("<td>"+lastName+ ... +"</td></tr>");
   }
   catch (AlreadyExistentException e)
   {
      // Already existing ...
      return(false);
   }
   return(true);
}
```

Figure 129. Sample Servlet Code for Creating a New Order

The numbers below refer to the numbers in Figure 129:

1 For *RRDS clusters*, the relative record number must be set before a new record can be added.

2 Adds the new order record to the cluster.

The coding described in Figure 129 will display this Web page:

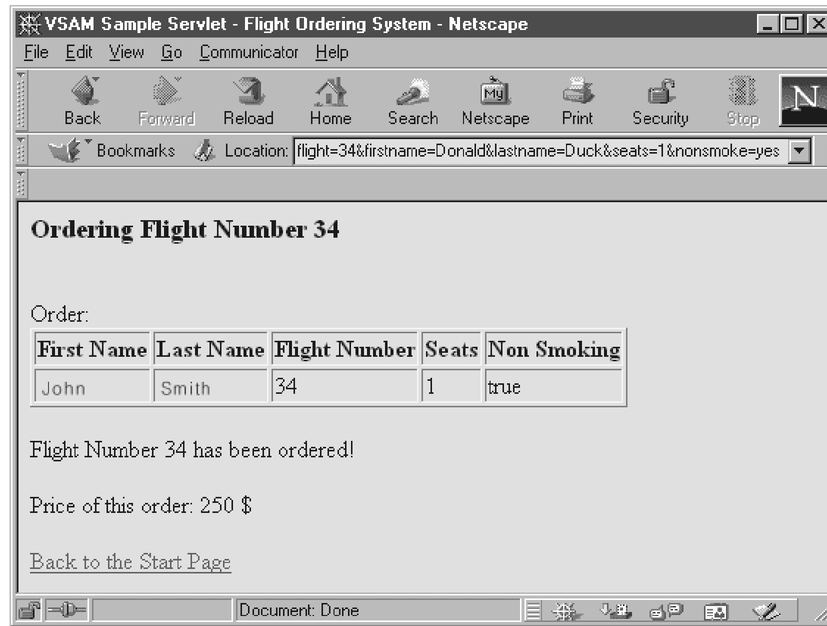


Figure 130. Flight Order Confirmation Window, Generated by the Sample Servlet

Chapter 19. Using Java Server Pages to Access Data

JSP Requests Java Server Pages (JSPs) are a way for developers who are familiar with HTML to easily create *servlets,* since JSPs are compiled into servlets. They are also useful for Java applications in which servlets, and other generators of dynamic HTML content, must be integrated with static HTML. The commonly-used naming conventions for JSPs is that the suffix ends with **.jsp**.

This chapter contains these main topics:

- "How JSPs Are Used in 3-Tier Environments"
- "Example of a Simple Java Server Page" on page 247

How JSPs Are Used in 3-Tier Environments

Figure 131 on page 246 describes how JSPs are used within the z/VSE *3-tier environment*. The number of each list item below describes a step shown in Figure 131 on page 246.

1 The client's Web browser sends the request for a JSP URL to the IBM HTTP Server on the physical/logical middle-tier. Each JSP is stored in the IBM HTTP Server's normal document hierarchy.

2 The IBM HTTP Server retrieves the JSP and sends it to the WebSphere Application Server.

3 The action now performed by the WebSphere Application Server depends upon whether or not this JSP has previously been compiled to a servlet:

- If the JSP has *not* previously been compiled to a servlet (or if the JSP has been modified since the last time it was used), the WebSphere Application Server uses its JSP-engine to do so.
- If the JSP *has* previously been compiled to a servlet (and has not been modified since the last time it was used), the resulting servlet will already be stored in the servlet repository. A compilation of the JSP is therefore not required.

The WebSphere Application Server then runs the servlet in its Java Virtual Machine (JVM). The servlet uses the VSE Java Beans class library (**VSEConnector.jar**) to build a connection to the VSE Connector Server.

4 The servlet sends the required HTML Web Page to the client's Web browser, via TCP/IP.

5 The client's Web browser displays the Web Page. The Web browser now sends a request for data to the servlet running in the WebSphere Application Server's Java Virtual Machine.

6 The servlet sends the request for data to the VSE Connector Server running on the z/VSE host (using the connection previously built using VSE Java Beans).

© Copyright IBM Corp. 2000, 2013

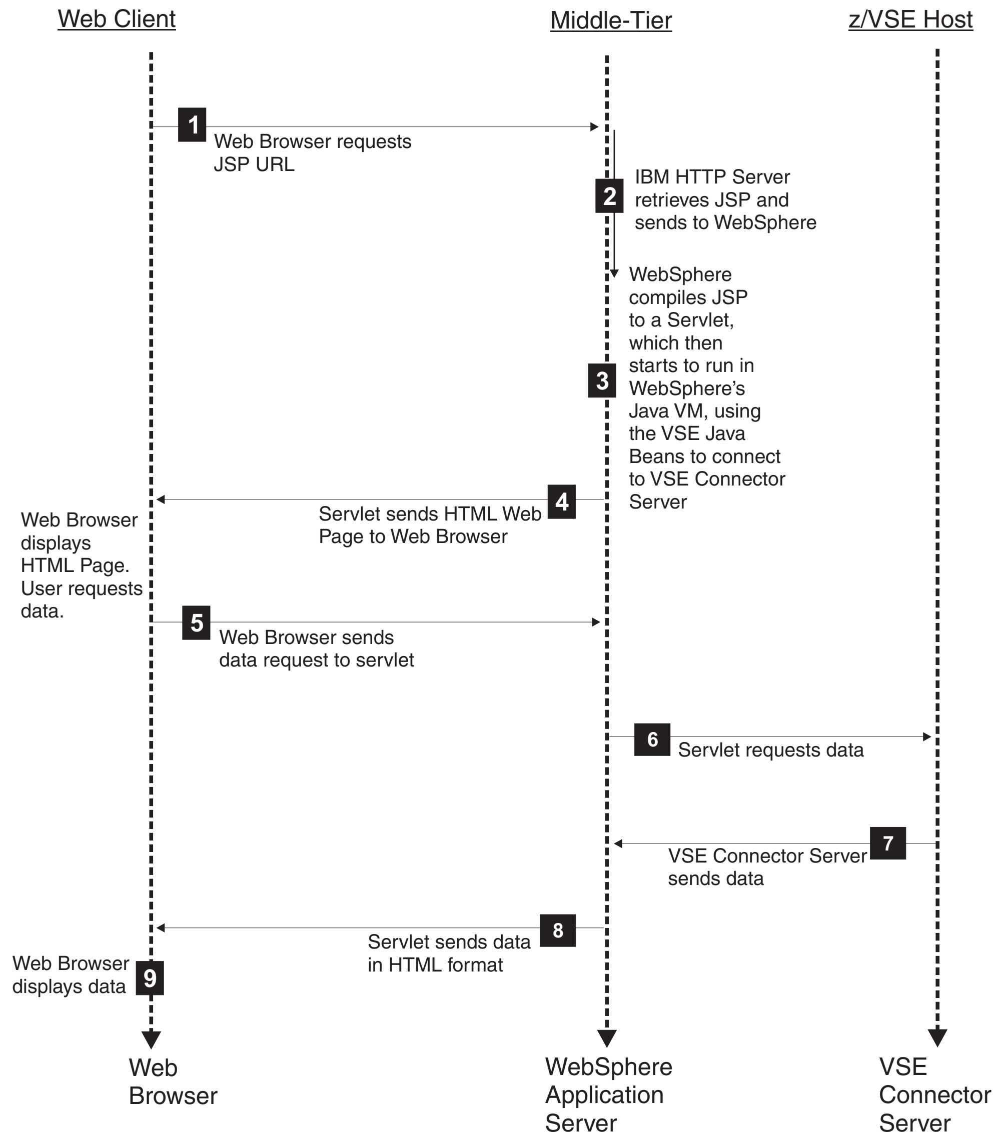


Figure 131. How JSPs Are Used in the z/VSE 3-Tier Environment

7 The VSE Connector Server obtains the data and sends it to the servlet.

Notes:

1. The VSE Connector Server could also be used for accessing POWER, VSE/ICCF, or Librarian, or sending console commands.
2. An alternate to using the *VSE Connector Server* is for accessing data is to use a DB2 Stored Procedure on the physical/logical middle-tier, which communicates directly with the VSAM file system on the z/VSE host.
3. If the end-user had requested DB2 data, the data could be retrieved using a DB2 Connect router on the physical/logical middle-tier and a DB2 Server for VSE on the z/VSE host. Alternatively, the DB2 data could be retrieved using a DB2 Connect router on the physical/logical middle-tier and a DB2 Stored Procedure on the z/VSE host.
4. If the end-user had requested DL/I data, the data could be retrieved using a user-written application on the physical/logical middle-tier which communicates directly with a DB2 Stored Procedure on z/VSE host, which can in turn accesses the DL/I databases of the z/VSE host.

5. If the end-user had requested CICS data, the data could be retrieved using an WebSphere MQ Server router on the physical/logical middle-tier and a WebSphere MQ Server for z/VSE on the z/VSE host.

8 The servlet generates a dynamic Web Page consisting of HTML code, and sends this together with the requested data back to the client's Web browser.

9 The client's Web browser re-displays the Web Page together with the requested data.

HTTP Sessions are used between the Web Client and the physical/logical middle-tier for sending and receiving data. Connect Sessions are used between the physical/logical middle-tier and the z/VSE host for sending and receiving data.

The advantage in using JSPs rather than servlets, is that by using JSPs your HTML has a wider functionality. You can create HTML from servlets, but you require a lot of programming effort. Using JSPs, however, you can carry out a change and then simply let the WebSphere Application Server recompile and execute a servlet which contains your changes.

JSPs also have the advantage that they can be used by authors who do not possess much HTML knowledge. A JSP normally accesses an Enterprise Java Bean (EJB) that encapsulates database queries and business logic. The data returned by the EJB is sent to the Web client in HTML format and can then be dynamically included into the current Web page.

Example of a Simple Java Server Page

Figure 132 shows how a JSP can be used to display the current date. These special tags (<% and %>) are used to enclose the Java code. The JSP engine dynamically compiles the JSP into a servlet. The servlet is then executed, and displays the Web page.

```
<html>
<head>
<title>My first JSP</title>
</head>
<% import java.util.*; %>
<% Date date = new Date();
   response.println("The current date is " + date);
   ...
%>
</body>
</html>
```

Figure 132. Example of a Java Server Page /JSP)

Chapter 20. Using EJBs to Represent Data

EJBs are “write once, run anywhere” distributed beans that run in a web application server environment, such as IBM's WebSphere Application Server.

EJBs are intended to represent either:
- data in a database (entity beans).
- the connection to a remote data store (session beans).

In a z/VSE 3-tier environment, you can use EJBs to represent:
- data in a relational database (DB2).
- non-relational data, such as DL/I or VSAM. This requires that you have previously mapped such non-relational data to a *relational* structure (for details, refer to Chapter 12, “Mapping VSE/VSAM Data to a Relational Structure,” on page 103).

EJBs are easy to implement. Furthermore, they:
- Are the most powerful feature of the WebSphere Application Server environment.
- Allow the application programmer to concentrate on developing the Java applications, without having to code the access to data.

This chapter contains these main topics:

- “Overview of the EJB Architecture”
- “Example of Using EJBs to Access VSAM Data” on page 254
- “Example of Implementing VSAM-Based EJBs” on page 256

Overview of the EJB Architecture

This information provides you with an overview of the EJB architecture. A practical example of how EJBs can be implemented, is provided in “Example of Implementing VSAM-Based EJBs” on page 256.

The WebSphere Application Server server must provide a number of services to EJBs in order to manage them properly. These services are provided by an entity called an *EJB Container* (described in “Overview of How EJB Containers are Used” on page 250).

There are two types of EJBs, *Entity beans* and *Session beans.* The table below shows the properties of Entity beans and Session beans.

Table 7. Properties of Session Beans and Entity Beans

Bean Type	Description	State and Persistence
Session	Short lived beans tied to a single client session	Two major forms of Session beans: **Stateless** Perform a simple operation. Holds no data. **Stateful** Maintains variables between client requests.

© Copyright IBM Corp. 2000, 2013

Table 7. Properties of Session Beans and Entity Beans (continued)

Bean Type	Description	State and Persistence
Entity	Long lived beans existing across many client sessions	Usually wrapped by Session beans. The EJB example provides two session beans that access one entity bean. All beans hold data, but there are two ways to manage persistence (synchronization with database): **Bean managed** Bean must get and put rows into the database. **Container managed** Bean gets and puts data automatically. The entity bean belonging to the EJB example implements *bean-managed* persistence.

Session beans perform the business logic in an application. These beans could be used to represent a shopping "cart" in an online ordering system, or to calculate sales tax on a purchase. A Session bean is a normal Java class tied to a single client session. They have several other restrictions. They cannot:

- Start new threads (since EJBs run inside the EJB Container and not within the Java Virtual Machine).
- Carry out any functions except representing one row of a database (VSAM record or DB2 row).
- Contain static read/write variables.
- Use **java.io** classes.

Entity beans have the following characteristic. They:

- must access data through JDBC or by some other means (since Entity beans directly represent data). The EJB example (included on page "Example of Implementing VSAM-Based EJBs" on page 256) uses the VSE Java Beans class library to access VSAM data on a remote z/VSE host.
- cannot be used without low-level support such as a JDBC interface or a connector to other data sources.
- can be used by several clients at the same time.
- have a lifetime equal to the length of time that the underlying data they represent, exists.

Overview of How EJB Containers are Used

EJB Containers act as an intermediaries between EJBs and clients, and also manage multiple EJB instances. After an EJB is written, it must be stored in a container which resides on the WebSphere Application Server (or another application server).

Figure 133 on page 251 shows how containers are used to manage EJBs.

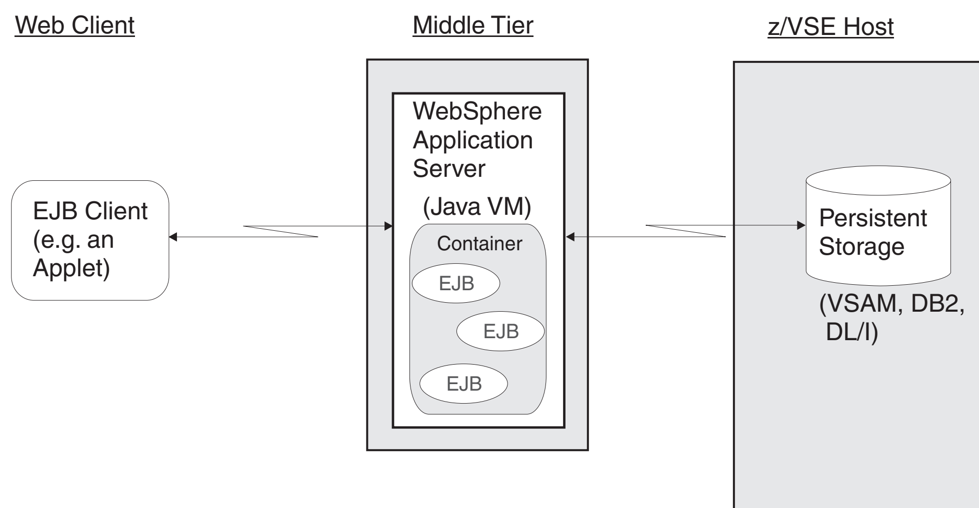


Figure 133. How Containers Are Used To Manage EJBs

Several tasks must be performed by the EJB container in order to fulfill its role as an EJB/client intermediary. These tasks include:

- Instance Passivation/Activation - Temporarily swapping EJBs in and out of storage.
- Instance Pooling - Sharing instances of EJBs among multiple clients.
- Database Connection Pooling - Allowing an EJB to use an open connection to the database from a pool of existing connections.
- Pre-cached Instances - Caching EJB state information to expedite the initial creation of EJBs.

Therefore, the container simplifies the EJB implementation process by managing all threading and client interaction with the EJBs. It also reduces the load on servers and databases by coordinating connection and instance pooling. As EJBs become standardized, vendors will increasingly provide EJB containers separately from application servers.

How EJBs Compare to JavaBeans / Java Servlets

The distinction between EJBs and JavaBeans lies primarily in their roles. JavaBeans are intended to be used as visual components in Graphical User Interfaces (GUIs), such as buttons and labels. On the other hand, EJBs are designed to either represent data in a database or perform actions on this data, and therefore have a restricted use.

Both EJBs and servlets reside in a multi-tier architecture. Servlets take in requests over HTTP, process the requests, and send responses over HTTP in the form of HTML code that can be viewed in a Web browser. Thus, servlets essentially provide User Interface functionality and business logic. When using EJBs, however, it is the *client's responsibility* to provide the User Interface.

Here is a summary of the limitations of using EJBs:

- They cannot access the local file system of the web server platform.
- There is no way to implement callback mechanisms, because EJBs are always single-thread programs. This is a disadvantage especially for long running actions, like all kind of requests that retrieve lists of objects.

- There is no way to stop a running action, nor to begin processing the result list before the complete action has finished.

Implementing Your Client Applications

Enterprise Java Beans allow you to separate the development of your client User-Interface from that of your business logic. Therefore, you can use various client platforms and multiple User Interfaces. EJBs can interact with many diverse client types including: other EJBs, Java applications, Java applets, Java servlets, and non-Java components by using CORBA connectivity.

Enterprise Java Beans provide easy connectivity to all Java and CORBA compatible components.

1. Using JNDI or another naming system, the client program locates the EJB home and uses the home to create a remote interface to access the bean.
2. This triggers the EJB container to create the bean.
3. After creating the remote interface, the bean appears to the client as any other class except that all passed objects must be serialized and sent over the network.

EJBs use RMI and 'stubs' to transmit information between the EJB container and the client computer. RMI requires that all objects passed across the network are serializable. Figure 134 shows the entities involved in a single EJB method call.

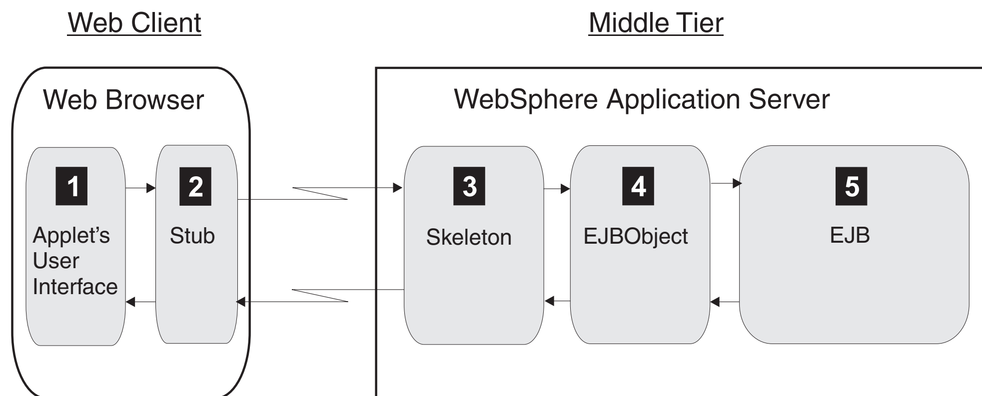


Figure 134. Overview of the Entities Involved in an EJB Method Call

The general processing during a method call is as follows:

1. The User Interface invokes the container generated stub, which serializes the arguments of the method call.
2. The generated stub sends the data over the network.
3. The skeleton stored on the EJB Server de-serializes the arguments, and passes them to the container-generated EJBObject (which is an implementation of the remote interface).
4. The EJBObject then interacts with the EJB.
5. When the EJB has completed its processing, it passes the result back to the EJBObject.

The reply is passed back to the User Interface, using the reverse path as described above. Therefore:

- The *server* manages calls to the EJBObject.
- The *container* manages calls to the EJB using the EJBObject.

EJBs connect to each other using the same method of calling, so it is very easy to implement a system of EJBs that communicate together. Even if the EJBs are in the same container, every method call must be made through the container to avoid multi-threading problems.

CORBA, Common Object Request Broker Architecture, provides a method for non-Java applications to connect to EJBs. CORBA works much the same way as RMI, but requires an application server that supports CORBA to EJB mapping and requires the use of different naming services.

How an EJB Client Accesses EJBs

Figure 135 shows how an EJB client accesses an Enterprise Java Bean (EJB). This information applies to both the example EJB and to EJBs in general.

The client never communicates directly with the EJB. Instead, it "talks to" the EJB via its *home interface* and its *remote interface*. The home interface and remote interface code are both created by the WebSphere Application Server during deployment of the EJBs.

The EJB server and EJB container are part of your Web application server (such as the WebSphere Application Server), and might be supplied by independent software vendors.

Therefore, this code must be implemented for an EJB:

- The *EJB client*, that communicates with the EJB. This is usually a servlet or an applet, but could also be any other Java program.
- The *EJB home interface*. The EJB developer simply specifies the interface. The EJB home interface is then created by the WebSphere Application Server when it deploys the EJB.
- The *EJB remote interface*. The EJB remote interface is also created by the WebSphere Application Server when it deploys the EJB.
- The *EJB class*. This class implements the EJB's business logic.

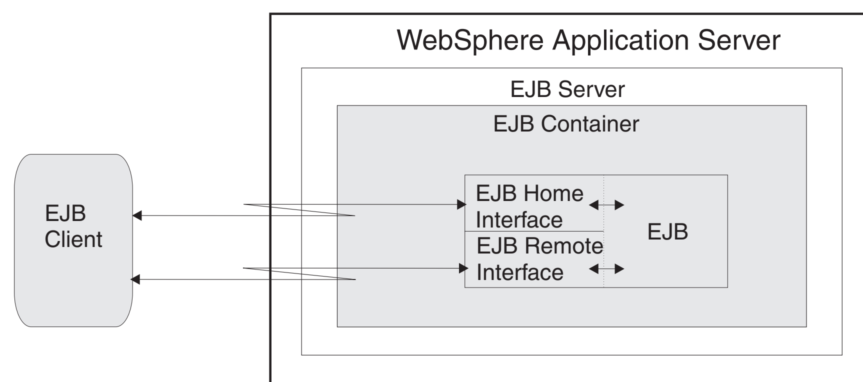


Figure 135. How an EJB Client Communicates with an EJB

In Figure 135, the *EJB client* can be:

Servlet or JSP
: Runs in the same WebSphere Application Server as the EJB. Using servlets as EJB clients makes it possible to obtain access to the EJB data from any

Web browser, via your company's Intranet or the Internet. The deployed **ejb-jar** file resides on the server that runs the WebSphere Application Server. For example, online shopping is often performed using a combination of static Web pages and servlets.

Java application
Runs on a different workstation and communicates with the EJB via a TCP/IP connection. You will normally use a Java application as an EJB client for test purposes in your development environment. A Java application is the most simple EJB client and can be used to implement a variety of test cases. Because the deployed **ejb-jar** file must be accessible by both the EJB client and the WebSphere Application Server, you must either copy the file to the workstation where your EJB client runs, or run the EJB client on the same machine as the WebSphere Application Server. You might also make the file accessible to both platforms over your company's network.

Applet
Runs in a Web browser on a different workstation and communicates with the EJB via a TCP/IP connection. If you require a more advanced user interface than is possible using HTML, you can use an applet as the EJB client. For example, online banking systems often use applets. In such systems, all classes that the applet requires, including the deployed **ejb-jar** file, are downloaded to the Web browser client when the applet runs. This can lead to performance problems, especially where the applet is accessed via the Internet.

Another EJB
Runs in the same or a different WebSphere Application Server as the EJB. For example, the employer bean and employee bean provided with the VSE Connector Client example, act as EJB clients when they access the record bean.

In other words, an EJB client is any Java program that accesses an EJB.

Example of Using EJBs to Access VSAM Data

Figure 136 on page 255 provides an illustration of how EJBs are used together with an applet in the z/VSE 3-tier environment, to access VSAM data:

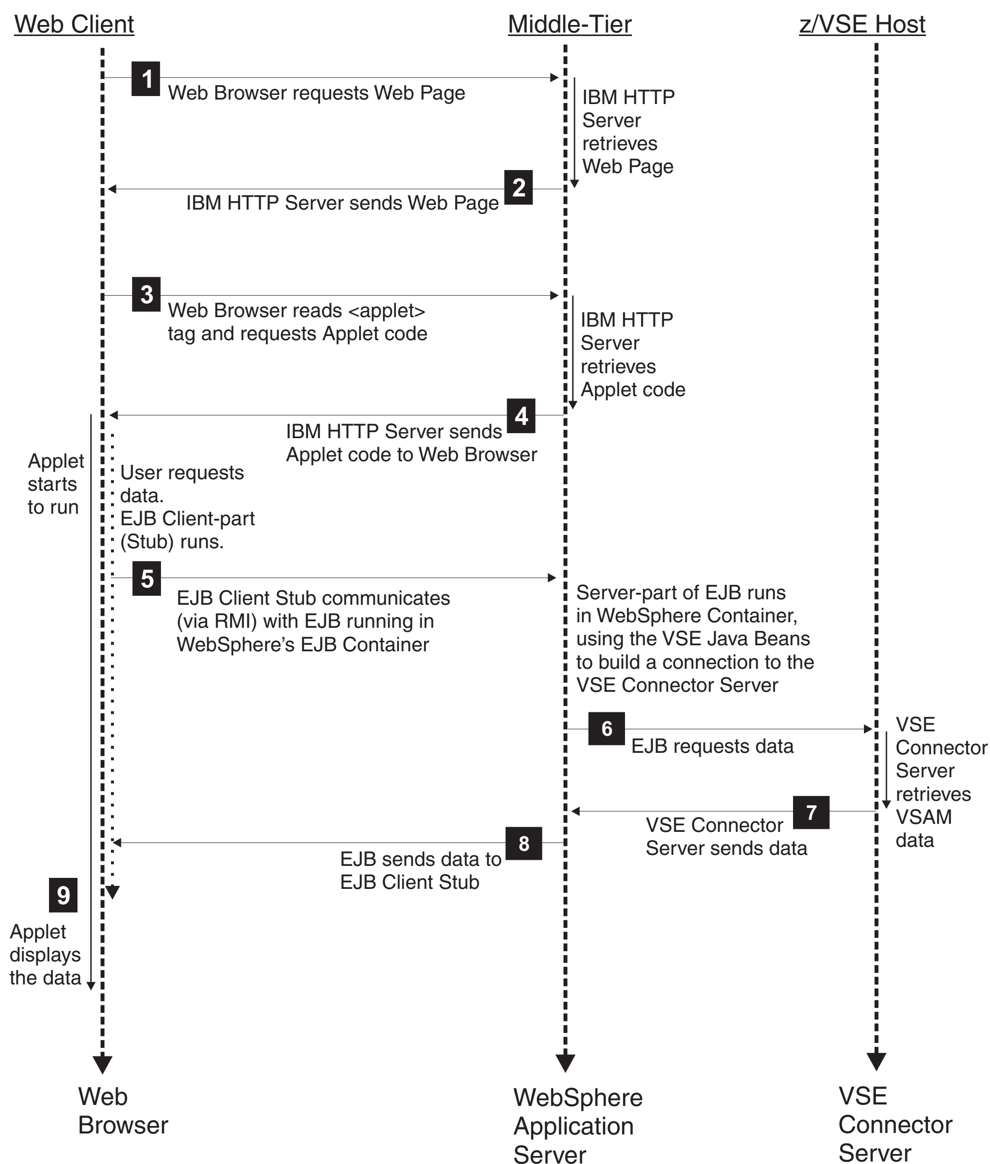


Figure 136. How EJBs Are Used Together with an Applet in the 3-Tier Environment

HTTP Sessions are used between the Web Client and the physical/logical middle-tier for sending and receiving data. Socket connections are used between the physical/logical middle-tier and the z/VSE host for sending and receiving data.

The number of each list item describes a step shown in Figure 136:

1 The client's Web browser sends a request for an HTML page to the IBM HTTP Server running on the physical/logical middle-tier. This example assumes that the HTML file contains an applet tag which then causes the applet to be called. The applet acts as an EJB client to access the EJB data.

2 The IBM HTTP Server retrieves the Web Page, and sends it to the client's Web browser.

3 The client's Web browser reads an <applet> tag, and sends a request for the applet code to the IBM HTTP Server running on the physical/logical

middle-tier. The IBM HTTP Server sends a *JAR* file (containing the applet and the Java routines required to run it) to the client's Web browser.

4 The IBM HTTP Server retrieves the applet code, and sends it to the client's Web browser.

5 The client's Web browser runs the applet. The end-user now requests data that is stored on the z/VSE host, via the use of EJBs that are stored on the physical/logical middle-tier. The applet uses EJB *stub classes* to communicate with the EJB server-part, via RMI. See Figure 133 on page 251 and Figure 134 on page 252 for further details. The server-part of the EJB uses the VSE Java Beans class library (**VSEConnector.jar**) to build a connection to the VSE Connector Server.

6 The server-part of the EJB requests the data from the VSE Connector Server running on the z/VSE host, using the connection previously built using the VSE Java Beans.

7 The VSE Connector Server obtains and then sends the requested VSAM data to the server-part of the EJB.

8 The server-part of the EJB sends the data to the EJB Stub running in the client's Web browser.

9 The applet running inside the client's Web browser displays the Web Page together with the requested data.

Example of Implementing VSAM-Based EJBs

EJBs are typically used so that one EJB instance encapsulates the data of one relational table row. The example provided here shows how to represent non-relational VSAM data, that has been mapped to a relational structure, by an EJB.

The mapping of non-relational data to a relational structure is described in Chapter 12, “Mapping VSE/VSAM Data to a Relational Structure,” on page 103.

This example consists of *three* EJBs:

- *RecordBean*, which is an entity bean, represents and contains a complete mapped VSAM record that belongs to a VSAM cluster containing data about employees.
- *EmployerBean* and *EmployeeBean*, which are session beans, provide the business logic to access the data represented by the *RecordBean*. They are used to implement two different views on this data, an employee's view and an employer's view.

In this example, utilities are also provided which are used to fill the VSAM cluster with sample data.

The execution of the sample EJB is described in these topics:

- “Step 1: Define the Sample's VSAM Cluster” on page 257
- “Step 2: Create the Record Layout for Employees” on page 257
- “Step 3: Specify the EJB's Home Interface” on page 258
- “Step 4: Specify the EJB's Remote Interface” on page 258
- “Step 5: Implement the RecordPK Class” on page 259
- “Step 6: Implement the EJB Code” on page 259
- “Step 7: Compile the Java Source Files” on page 264

- "Step 8: Deploy the EJBs" on page 264
- "Step 9: Access the EJBs from an EJB Client" on page 265

Step 1: Define the Sample's VSAM Cluster

In this first step, the VSAM cluster that is used by this example is created, and filled with sample data.

Note: Because the column placement of the label information in the IESVCLUP step is critical, you are recommended to use the z/VSE Interactive Interface dialogs to define this file!.

Use the following job to define the sample cluster EJB.VSAM.EXAMPLE:

```
* $$ JOB JNM=DEFINE,CLASS=0,DISP=D,NTFY=YES
// JOB  DEFINE EJB SAMPLE CLUSTER
// EXEC IDCAMS,SIZE=AUTO
 DEFINE CLUSTER ( -
        NAME (EJB.VSAM.EXAMPLE                            ) -
        CYLINDERS(2       2         ) -
        SHAREOPTIONS (2) -
        RECORDSIZE (80     80    ) -
        VOLUMES (DOSRES ) -
        NOREUSE -
        INDEXED -
        FREESPACE (15 7) -
        KEYS (4   0     ) -
        NOCOMPRESSED -
        TO (99366  )) -
        DATA (NAME (EJB.VSAM.EXAMPLE.@D@                        ) -
        CONTROLINTERVALSIZE (4096  )) -
        INDEX (NAME (EJB.VSAM.EXAMPLE.@I@                       )) -
        CATALOG (VSESP.USER.CATALOG                         )
  IF LASTCC NE 0 THEN CANCEL JOB
/*
// OPTION STDLABEL=ADD
// DLBL EJBSAMP,'EJB.VSAM.EXAMPLE',,VSAM,                             X
               CAT=VSESPUC
/*
// EXEC IESVCLUP,SIZE=AUTO
A EJB.VSAM.EXAMPLE                            EJBSAMP VSESPUC
/*
/&
* $$ EOJ
```

Step 2: Create the Record Layout for Employees

Create the record layout for employees, using this layout:

```
EMPNUM     Unsigned, offset=0,  length=4   (employee number = key)
PASSWORD   String,   offset=4,  length=10
NAME       String,   offset=14, length=25
DEPT       Unsigned, offset=39, length=4   ("department number")
HOURLY     Unsigned, offset=43, length=4   ("hourly wage")
PTD        Unsigned, offset=47, length=4   ("paid to day")
TNP        Unsigned, offset=51, length=4   ("time not paid")
```

- *To access the record of a specific employee*, you must specify the employee's number and a password.
- *To add new data to the cluster*, use the utility program *EJBPrepareData.java*. This program is supplied with the VSE Connector Client.
- *To create the map and its data fields*, use utility program *EJBPrepareData.java*.
- *To display the sample data*, use the Java program *EJBShowData.java*, which is supplied with the VSE Connector Client. You can also view the data using the

VSE Navigator application, which can be downloaded from the VSE Homepage. For details of how to access the VSE Homepage, see "Where to Find More Information" on page xix.

- *To run the pre-compiled utilities*, go to the **\<install-directory>\samples** directory and enter:

```
java com.ibm.vse.ejb.vsamexample.EJBPrepareData
java com.ibm.vse.ejb.vsamexample.EJBShowData
```

Step 3: Specify the EJB's Home Interface

In this step, the *EJB home interface* (described in Figure 135 on page 253) is specified for the *RecordBean*. For details of how to specify the home interface for other EJBs, refer to the VSE Connector Client online documentation (see "Using the Online Documentation Options" on page 26 for details).

The EJB's home interface inherits its methods from the interface *EJBHome*. It must define method signatures for:

- Any *create()* methods.
- The *findByPrimaryKey()* method.
- Other "finder" methods.

When an EJB is deployed, the WebSphere Application Server creates the home interface code, together with the remote interface and the service stub class, that are used to access the EJB.

The method signatures are implemented in the source file *RecordHome.java* as follows:

```
package com.ibm.vse.ejb.vsamexample;
import javax.ejb.*;
import java.rmi.*;

public interface RecordHome extends EJBHome
{
   public Record create(int empnum, String name, String password, int dept,
                        int hourly, int paytodate, int hoursnotpaid)
   throws java.rmi.RemoteException, javax.ejb.CreateException;

   public Record findByPrimaryKey(RecordPK pk)
   throws RemoteException, FinderException;
}
```

Step 4: Specify the EJB's Remote Interface

In this step, the *EJB remote interface* (described in Figure 135 on page 253) is specified for the *RecordBean*. For details of how to specify the remote interface for other EJBs, refer to the VSE Connector Client online documentation (see "Using the Online Documentation Options" on page 26 for details).

When an EJB is deployed, the WebSphere Application Server creates the remote interface code, together with the Home Interface and the service stub class, that are used to access the EJB.

The remote interface must provide:

- Inheritance from the interface *EJBObject*.
- Method signatures related to the EJB's business methods
  - The code for these methods is generated during the process of deploying the bean.

– The method signatures are provided in the source file *Record.java*, as shown below.

```
package com.ibm.vse.ejb.vsamexample;
import javax.ejb.*;
import java.rmi.*;
import java.io.*;
import java.util.*;

public interface Record extends EJBObject
{
   public int getEMPNum() throws RemoteException;
   public String getPassword() throws RemoteException;
   public String getName() throws RemoteException;
   public int getDept() throws RemoteException;
   public int getHourly() throws RemoteException;
   public int getPayToDate() throws RemoteException;
   public int getTimeNotPaid() throws RemoteException;
   public void setDept(int dept) throws RemoteException;
   public void setEMPNum(int empnum) throws RemoteException;
   public void setName(String name) throws RemoteException;
   public void setHourly(int hourly) throws RemoteException;
   public void setPassword(String passwd) throws RemoteException;
   public void setPayToDate(int paytodate) throws RemoteException;
   public void setTimeNotPaid(int timenotpaid) throws RemoteException;
}
```

Step 5: Implement the RecordPK Class

The *RecordPK* class is a helper class, required by entity beans to provide objects that implement *java.io.Serializable*. Only serializable objects may be passed as arguments or as return values over RMI.

Serializable objects are returned, for example, by *ejbCreate()* or *ejbFindByPrimaryKey()*.

The code is implemented in source file **RecordPK.java**. You must specify the primary key class for entity beans, during the EJB deployment process (as described in "Step 8: Deploy the EJBs" on page 264).

```
package com.ibm.vse.ejb.vsamexample;

public class RecordPK implements java.io.Serializable
{
   public int empnum;
}
```

Step 6: Implement the EJB Code

The code provided here implements the sample EJB *RecordBean*. Methods are provided to:

- Access the *RecordBean*'s data.
- Connect to the remote z/VSE host.
- Obtain data from the database.
- Update the database.

This topic includes only parts of the code. If you require the complete source code, refer to the file **RecordBean.java** provided with the VSE Connector Client (see "Using the Online Documentation Options" on page 26 for details).

```
...
public class RecordBean implements EntityBean
{
```

```
/* the indexes of the fields in the backend map */
private int EMPNUM_INDEX = 0;
private int NAME_INDEX = 2;
private int PASSWORD_INDEX = 1;
private int DEPT_INDEX = 3;
private int HOURLY_INDEX = 4;
private int PAYTODATE_INDEX = 5;
private int TIMENOTPAID_INDEX = 6;

private transient VSESystem system;
private transient VSEVsamRecord record;

private transient EntityContext ctx;
private int empnum = 0;
...
```

Step 6.1: Implement the Methods of the EntityBean Interface

The methods described here implement the methods of interface *EntityBean*. These methods:

1. Access the remote database.
2. Fill the given EJB instance with one VSAM record.

The example uses the *ejbFindByprimaryKey()* method, because the EJB has been implemented using *bean-managed* persistence[3]. As a result, the EJB implements all the database calls that are required to send the object out to the database, and to read it back in again.

The logic for accessing the database is implemented in methods *ejbLoad()* and *ejbStore()*. The *ejbCreate()* method creates a new employee record with the parameters shown below, and returns its primary key.

```
public RecordPK ejbCreate(int empnum, String name, String password,
                          int dept, int hourly, int paytodate,
                          int hoursnotpaid)
throws CreateException
{
   RecordPK rpk= new RecordPK();
   rpk.empnum = empnum;
   try {
      rpk = this.ejbFindByPrimaryKey(rpk);
   }
   catch (Exception e)
   {
      try {
         system = connectVSE();
         record = new VSEVsamRecord(system, "VSESP.USER.CATALOG",
                                    "EJB.VSAM.EXAMPLE", "EJBMAP");
         record.setKeyField(EMPNUM_INDEX, new Integer(empnum));
         record.setField(NAME_INDEX, name);
         record.setField(PASSWORD_INDEX, password);
         record.setField(DEPT_INDEX, new Integer(dept));
         record.setField(HOURLY_INDEX, new Integer(hourly));
         record.setField(PAYTODATE_INDEX, new Integer(paytodate));
         record.setField(TIMENOTPAID_INDEX, new Integer(hoursnotpaid));
         record.add();
         rpk.empnum = empnum;
         this.empnum = empnum;
         return (rpk);
```

3. The opposite of bean-managed persistence is called *container-managed* persistence. Here, the EJB developer does not have to bother with synchronizing with the database. Instead, the entity bean's deployment descriptor specifies the fields that should be managed by the EJB container. At runtime, the container calls the *ejbLoad()* and *ejbStore()* methods when required, but the EJB developer must not provide any code for these methods.

```
         }
         catch (IOException ioe)
         {
            ...
         }
      }
   }
```

The following three methods must be implemented, because the EJB implements *bean-managed* persistence:

- ejbLoad()
- ejbStore()
- ejbFindByPrimaryKey()

```
   /**
    * fill the EJB with new data from the remote database.
    */
   public void ejbLoad() throws RemoteException
   {
      RecordPK pk = (RecordPK) ctx.getPrimaryKey();
      system = connectVSE();
      try {
         record = findRecord(pk.empnum, system);
      }
      catch (FinderException e)
      {
         ...
      }
   }

   /**
    * make a change permanent in the remote database.
    */
   public void ejbStore() throws RemoteException
   {
      try {
         record.commit();
      }
      catch (IOException e)
      {
         ...
      }
   }

   /**
    * looks up the record and returns pk back to the caller.
    * If the record is not found, a FinderException is thrown
    * We have to provide this method, because we implement the EJB
    * using bean-managed persistence.
    */
   public RecordPK ejbFindByPrimaryKey(RecordPK pk)
   throws FinderException, RemoteException
   {
      VSESystem system = connectVSE();
      VSEVsamRecord record = findRecord(pk.empnum, system);
      return (pk);
   }
```

The methods described here are also contained in the interface *EntityBean*. There are a number of other interface methods that are not shown here: if you require the complete source code, refer to the VSE Connector Client online documentation (see “Using the Online Documentation Options” on page 26 for details).

Note: The *ejbRemove()* method only deletes the local instance of the VSAM record that is related to this EJB. To commit the change in the remote database, a call to *ejbStore()* is required.

```
public void ejbRemove() throws RemoteException
{
   try {
      record.delete();
   }
   catch (IOException e)
   {
      throw new RemoteException("" + e);
   }
}

public void ejbActivate () throws RemoteException
{
   system = connectVSE();
   try {
      findRecord(empnum, system);
   }
   catch (FinderException e)
   {
      throw new RemoteException("FinderException");
   }
}
```

Step 6.2: Access z/VSE Host and Get Records from the Database

In this topic, the methods are implemented to:

- Access the z/VSE host.
- Get records from the database.

```
/**
  * Create connection specification. The connection spec holds
  * information about the physical host connection.
  */
 public VSESystem connectVSE() throws RemoteException
 {
    VSEConnectionSpec spec;
    VSESystem system;
    try
    {
       spec = new VSEConnectionSpec(InetAddress.getByName("9.164.155.95"),
                                    2893, "sysa","mypassw");

       // This is application server dependent
       Properties p = new Properties();
       p.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
             "com.ibm.websphere.naming.WsnInitialContextFactory");
       p.put(javax.naming.Context.PROVIDER_URL, "iiop:///");

       Context ctx = new InitialContext(p);
       spec.setJNDIContext(ctx);
       spec.setJNDIName("eis/VSEConnector");

       /* Stay logged on with this user for lifetime of this connection */
       spec.setLogonMode(true);

       /* Create VSE system instance with this connection */
       system = new VSESystem(spec);
    }
    catch (java.net.UnknownHostException e)
    {
       throw new RemoteException("Unknown host");
    }
 }
```

```
/**
 * find and load the record with pk empnum from VSAM.
 */
protected VSEVsamRecord findRecord(int empnum, VSESystem system)
throws FinderException
{
   try {
      VSEVsamRecord rec = new VSEVsamRecord(system, "VSESP.USER.CATALOG",
                                   "EJB.VSAM.EXAMPLE", "EJBMAP");
      rec.setKeyField(EMPNUM_INDEX, new Integer(empnum));
      if (rec.isExistent())
      {
         rec.refresh();
         return rec;
      }
      else
      {
         throw new FinderException("Record not found.");
      }
   }
   catch (IOException e)
   {
      throw new FinderException("IOException");
   }
}
```

Step 6.3: Implement the Set & Get Methods to Access the Data Fields

In this topic, the set and get methods are implemented in order to access the data fields (columns). These methods are the interface that EJB clients use, in order to access the encapsulated data of the EJB.

For each column you wish to access, you must provide a *get* method.

For all columns that are to be updated, you must provide a *set* method.

Notes:

1. These methods do not access the remote database. Instead, they simply return or modify the EJB's internal data.
2. If you want to *permanently* update changes, you must use the interface method *ejbStore()*.

```
/**
 * returns the employee number for the record.
 */
public int getEMPNum() throws RemoteException
{
   try {
      return ((Integer)record.getField(EMPNUM_INDEX)).intValue();
   }
   catch (IOException e)
   {
      throw new RemoteException("IOException");
   }
}


/**
 * set the employee number for the current record
 */
public void setEMPNum(int empnum) throws RemoteException
{
   try
```

```
        {
           record.setField(EMPNUM_INDEX, new Integer(empnum));
        }
        catch (IOException e)
        {
           throw new RemoteException("IOException");
        }
        return;
     }
     ...
  }
```

Step 7: Compile the Java Source Files

You must compile the Java source code described in this topic, in order to set up the EJB sample. Run the compile jobs from directory **\<install-directory>\samples**.

Notes: You must include certain JAR files containing EJB-related classes, in your local classpath. For details, refer to the:
- VSE Connector Client online documentation (see "Using the Online Documentation Options" on page 26 for details).
- The documentation references provided within "Installing the WebSphere Application Server" on page 21.

The VSE Connector Client provides already-compiled class files, for all sample Java source files. However, the following source files belong to the EJB example.

1. Job to compile the utilities to create and display the sample data:
   ```
   javac com\ibm\vse\samples\EJBPrepareData.java
   javac com\ibm\vse\samples\EJBShowData.java
   ```
2. Job to compile the EJB source files:
   ```
   javac com\ibm\vse\samples\Employer.java
   javac com\ibm\vse\samples\EmployerHome.java
   javac com\ibm\vse\samples\EmployerBean.java
   javac com\ibm\vse\samples\Employee.java
   javac com\ibm\vse\samples\EmployeeHome.java
   javac com\ibm\vse\samples\EmployeeBean.java
   javac com\ibm\vse\samples\Record.java
   javac com\ibm\vse\samples\RecordHome.java
   javac com\ibm\vse\samples\RecordBean.java
   ```
3. Job to compile the EJB clients:
   ```
   javac com\ibm\vse\samples\EJBApplet.java
   javac com\ibm\vse\samples\EJBClient.java
   ```

Step 8: Deploy the EJBs

The method you use to deploy your EJBs differs according to the version of the WebSphere Application Server you are using. The VSE Connector Client online documentation provides detailed instructions on how to deploy EJBs for various versions of the WebSphere Application Server (see "Using the Online Documentation Options" on page 26).

For details of how to deploy EJBs:
1. Proceed to the main window of the Online Documentation, as described in "Using the Online Documentation Options" on page 26.
2. Select **Programming Concepts**.
3. Select **EJBs**.
4. Select **An example to represent VSAM records**.

5. Select either:
   - **Deploy the EJBs on WebSphere 3.x**
   - **Deploy the EJBs on WebSphere 4.x**

Step 9: Access the EJBs from an EJB Client

Prerequisites for Accessing the EJBs from an EJB Client

Before you can begin accessing the EJBs you have created from an EJB client, these conditions must be met:

1. You have successfully created the VSAM cluster required for running the sample, as described in "Step 1: Define the Sample's VSAM Cluster" on page 257.
2. You have filled the above cluster with sample data, for example by using the utility *EJBPrepareData*, as described in "Step 2: Create the Record Layout for Employees" on page 257.
3. TCP/IP for VSE/ESA must be running on the z/VSE host (as described in "Configuring and Activating TCP/IP for VSE/ESA" on page 19).
4. The VSE Connector Server must be running on the z/VSE host (as described in "Starting the VSE Connector Server" on page 34).
5. You have specified the correct IP address, a VSE user ID, and password, in **RecordBean.java** (as described in "Using the Online Documentation Options" on page 26).
6. The IBM HTTP Server must be running on the physical/logical middle-tier (as described in "Configuring and Activating the VSE HTTP Server" on page 19).
7. The WebSphere Application Server must be running on the physical/logical middle-tier (refer to the documentation references provided in "Installing the WebSphere Application Server" on page 21).
8. The sample EJBs are running (as described in "Step 8: Deploy the EJBs" on page 264).

How EJBs Are Accessed from an EJB Client

An EJB client accesses the business logic contained in the EJBs, in this general way:

1. The EJB client uses a naming service to locate the EJB's home interface.
2. The naming service (usually the Java Naming and Directory Interface, JNDI) returns a reference to an object that implements the EJB's *home interface* (described in "How an EJB Client Accesses EJBs" on page 253).
3. The EJB client makes a call on the EJB's home interface, to gain access to the EJB's *remote interface* (described in "How an EJB Client Accesses EJBs" on page 253).
4. The EJB client make calls to the EJB's business methods against the remote interface.

However, the actual code you use to access an EJB from an EJB client will vary according to the type of Web Application Server you are using.

Figure 137 on page 266 shows how the *EJBClient* sample application accesses the EJBs via their home interfaces (shown as **H**), and remote interfaces (shown as **R**). If you wish to see the complete code for accessing the EJBs using the EJB Client, refer to the VSE Connector Client online documentation.

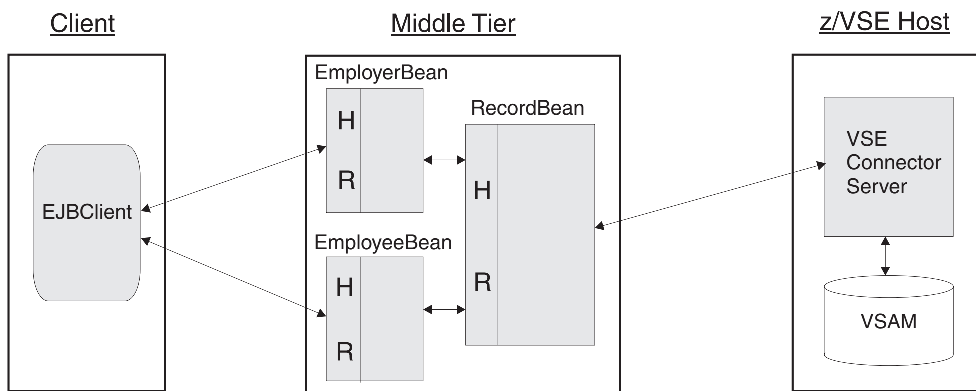


Figure 137. How the EJB Client Accesses EJBs in the Provided Example

The two session beans are in fact themselves EJB clients. When accessing the *RecordBean*, they perform the same processing of looking up the home interface and remote interface, as the EJB client does when it gets access to the session beans.

The session beans implement two different views on the same data, by including or not a particular column into a view. In addition, however, the session beans specify access rights for specific columns. For example, the *EmployeeBean* might only be able to read a given column, whereas the *EmployerBean* might also be able to update this column.

Sample EJB Client Source Code for Accessing EJBs from an EJB Client

This topic shows the source code for the sample EJBClient application, that is part of the VSE Connector Client. The code is based upon the use of the *WebSphere Application Server* as the Web Application Server type.

```
public class EJBClient
{
   public static void main(String[] argv)
   {
      try
      {
         EmployerHome employerh;
         Properties p = new Properties();  1
         p.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
              "com.ibm.websphere.naming.WsnInitialContextFactory");
         p.put(javax.naming.Context.PROVIDER_URL, "iiop:///");
         InitialContext ic = new InitialContext(p);  2
         Object obj = (Object) ic.lookup("EmployerBean");  3
         if (obj instanceof org.omg.CORBA.Object)  4
         {
            employerh =
                (EmployerHome)(javax.rmi.PortableRemoteObject.narrow(
                         (org.omg.CORBA.Object)obj, EmployerHome.class));
         }
         else
         {
            System.out.println(
                   "Did not get an org.omg.CORBA.Object from lookup().");
            return;
         }
         Employer employer = employerh.create(1003, "newpass3");  5
         System.out.println("employer.calcPayToDate() = " +  6
                                    employer.calcPayToDate());
         Vector v = employer.getEMPInfo();
         System.out.println("employer.getEMPInfo() :");
         for (int i=0;i<v.size();i++)
            System.out.println("  " + v.elementAt(i).toString());
      }
      catch (Exception e)
      {
         ...
      }
   }
}
```

Figure 138. Example of EJB Client Code

The numbers below refer to the numbers in Figure 138:

1 Define properties. Specifying **iiop:///** as the PROVIDER_URL causes the EJB client to search for a name server on the local host listening on port number 900, which is the default for the WebSphere Application Server. In a real environment you would specify a *complete* URL, such as:

```
iiop://bankserver.mybank.com:9019
```

2 The initial context object is instantiated.

3 The home interface of the EmployerBean is looked up. The string used here to identify the EJB, is stored in the EJB's deployment descriptor. You can also view the EJB's deployment properties by using the *WebSphere Administrative Console*.

4 Cast the returned object to the EJB's home interface class. After an object is returned by the lookup method, you must use the static method *PortableRemoteObject.narrow()* to obtain an EJB home object for the specified EJB.

5 Create a remote interface object to obtain access to the EJB's business logic methods. This call to the remote interface *create()* method, invokes the *ejbCreate()* method of the EJB.

6 Access the EJB's business logic methods. For example, get some properties of employer that has the number 1003. Please be aware however, that the EJB is always accessed via the *remote* object (in this example, “Employer.class”). An EJB client cannot call methods that belong to the EJB directly (in this example, the **EmployerBean.class**). The above example employee number and password are taken from the EJB sample provided with the VSE Connector Client, which also provides utilities to fill the sample VSAM cluster with some sample data.

Chapter 21. Extending the Java-Based Connector

This chapter describes how you can extend the Java-based connector in 2-tier and 3-tier environments, by writing your own "plugins". It makes much use of the "Date and Time plugin" example, which obtains and displays a z/VSE host's current date and time.

By writing your own plugins, you can provide additional access to:

- *resources* (such as accessing VSAM files that are currently open in CICS).
- *applications* (such as starting CICS transactions, or starting Vendor applications).

You can write your plugin to consist of:

- A *client plugin* only. You do so by extending the VSE Java Beans class library. In this case, your plugin is a JavaBean which itself uses the (existing) JavaBeans to access data or provide services. **Note:** This JavaBean does *not* implement *VSEPlugin* (see below for details).
- A *client plugin* and a *server plugin*. In this case, your server plugin extends the capabilities of the VSE Connector Server by providing additional functionality that is required by your client plugin.

Your server plugin consists of a VSE PHASE that is a plugin to the VSE Connector Server. This PHASE (which you write using LE/VSE-C) is loaded during startup of the VSE Connector Server. The PHASE must also provide a well-defined interface to enable the VSE Connector Server to call the implemented functions.

Your client plugin consists of one or more JavaBeans written by yourself. These JavaBeans implement the Java class *VSEPlugin*, which is contained in the package **com.ibm.vse.connector**. For further details of how you can write your own JavaBeans, refer to the online documentation.

This chapter contains these main topics:

- "Implementing a Server Plugin"
- "Implementing a Client Plugin" on page 286
- "General Considerations When Designing Your Plugin" on page 288

Implementing a Server Plugin

A server plugin consists of a set of *callback functions* that are called by the VSE Connector Server in a given sequence. Each server plugin consists of a PHASE that you write using LE/VSE-C, which is loaded during VSE Connector Server startup. Your plugin must provide a well-defined interface to allow the VSE Connector Server to call the functions implemented in your server plugin. These callback functions (PluginMainEntryPoint, SetupPlugin, and so on) are described later in this topic.

An overview of how the functions are called by the VSE Connector Server, is provided in Figure 139 on page 270:

© Copyright IBM Corp. 2000, 2013

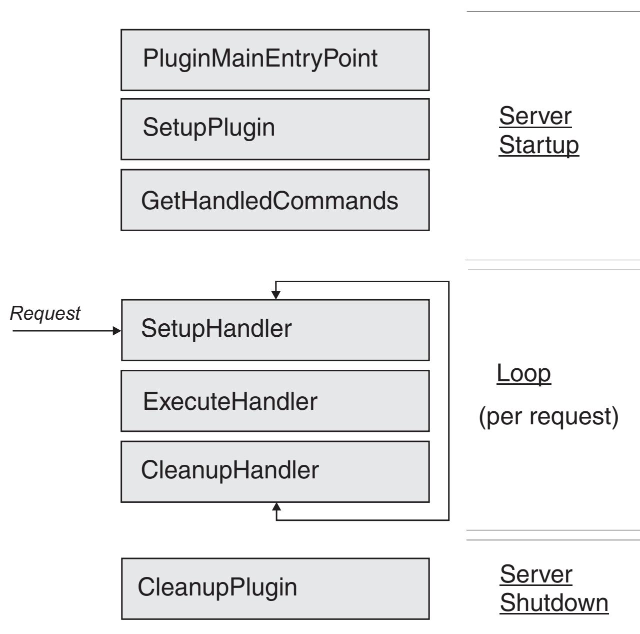


Figure 139. Overview of How a Server Plugin's Functions Are Called

A *typical sequence* of how functions are called by the VSE Connector Server, is provided in Figure 140 on page 271, Figure 141 on page 272, and Figure 142 on page 273:

- Figure 140 on page 271 describes the server plugin's functions that are called during the startup of the VSE Connector Server.
- Figure 141 on page 272 and Figure 142 on page 273 describe the server plugin's functions that are called when the VSE Connector Server receives a request from a client, and after the request processing is finished, during shutdown of the VSE Connector Server.

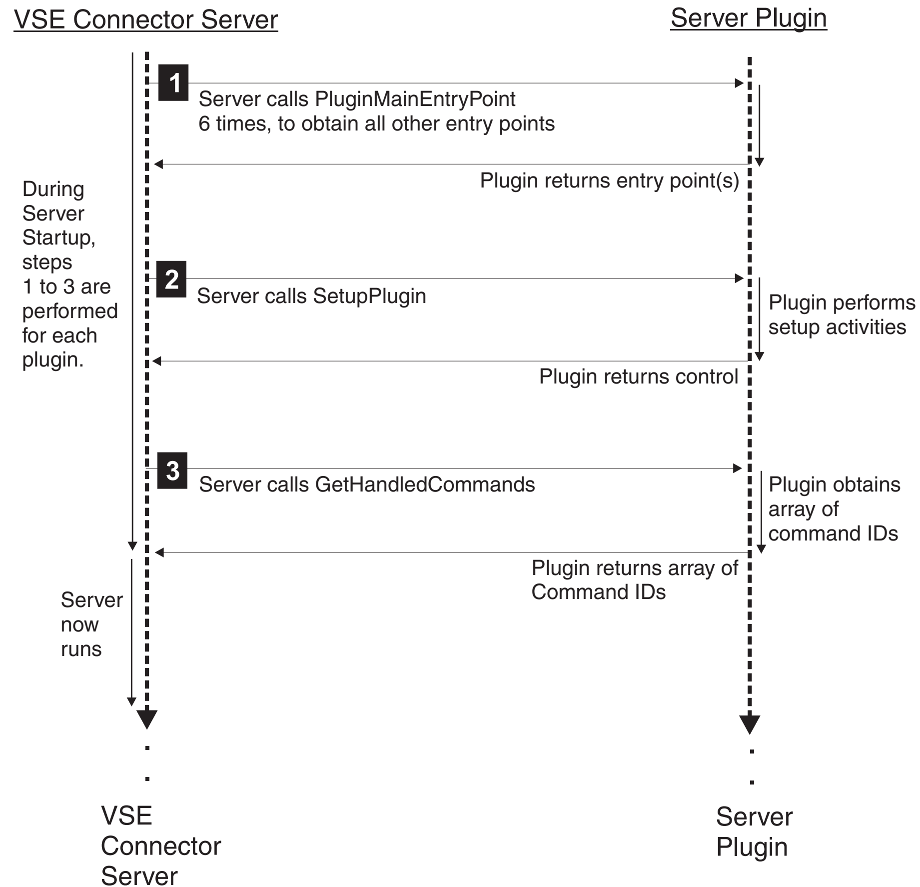


Figure 140. How a Plugin's Functions Are Called During Startup

1 By loading the server plugin phase (CDLOAD), the VSE Connector Server obtains the entry point of the server plugin (*PluginMainEntryPoint*). This entry point is then called six times to obtain the entry points of the other server plugin functions. Each call returns the entry point of a function specified by the parameter *iFuncID*. When this step is completed, the server has a list of the entry points in the plugin phase.

2 The *SetupPlugin* function is the first function that the VSE Connector Server calls, after the server plugin has been loaded. This function is used to initialize all request-independent resources (allocate storage, open database connections, and so on).

3 The VSE Connector Server calls the *GetHandledCommands* function immediately after calling the *SetupPlugin* function. *GetHandledCommands* returns a list of IDs that are accepted by the server plugin. The VSE Connector Server later uses this information in order to direct a request to a particular server plugin. Therefore, command IDs must be unique across all server plugins. The VSE Connector Server accepts a predefined range of command IDs. Refer to the header file IESPLGIN.H file for details.

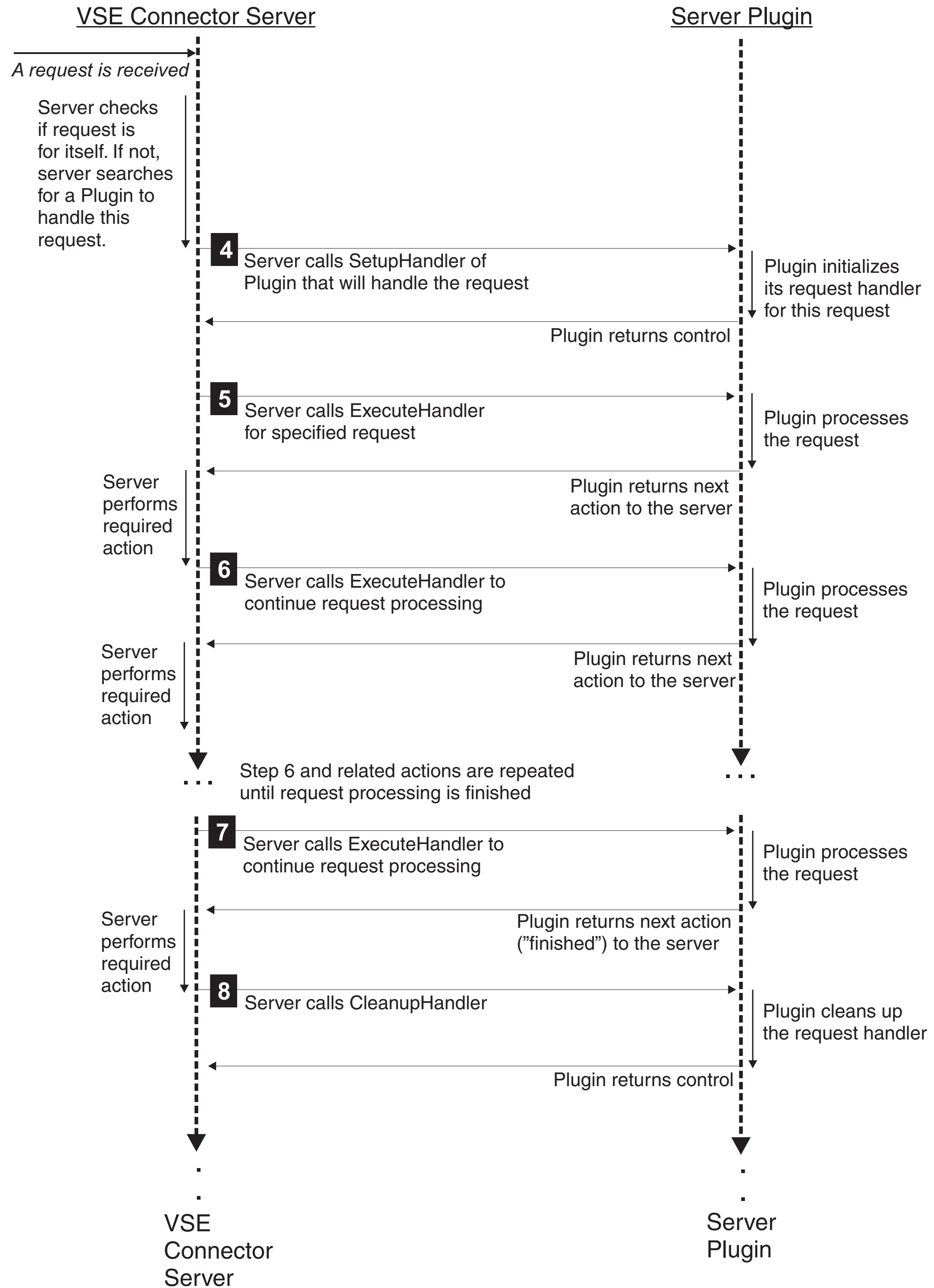


Figure 141. How a Plugin's Functions Are Called When a Request Is Received

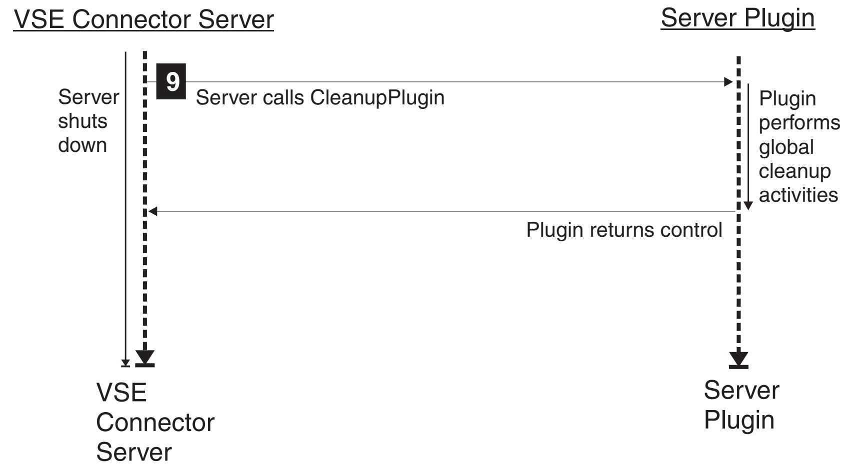


Figure 142. Overview of How a Plugin's Functions Are Called During Server Shutdown

4 After a request has been received from a client, the VSE Connector Server first determines which server plugin should handle the request, and calls the *SetupHandler* function of the selected server plugin. The server plugin allocates a *request handler* for this particular request, consisting of a control block used to process the request. The control block is allocated by the *SetupHandler* function and is specific to a request.

5 To process the specific request, the VSE Connector Server calls the *ExecuteHandler* function of the server plugin. The server plugin normally splits this code into small chunks, each of which is called by the VSE Connector Server during the next calls to the *ExecuteHandler* function (steps '6' and '7' below). The control block allocated by the *SetupHandler* function is passed for each call to the *ExecuteHandler* function. This control block can also be used to store local variables. The VSE Connector Server then processes the action returned by the *ExecuteHandler* function, which can be sending or receiving data, or waiting for timers or ECBs.

6 The VSE Connector Server again calls the *ExecuteHandler* function of the server plugin, to continue processing the request. The control block allocated by the *SetupHandler* function is again passed to the *ExecuteHandler* function. The VSE Connector Server then processes the action returned by the *ExecuteHandler* function.

7 The VSE Connector Server calls the *ExecuteHandler* function of the server plugin, to continue processing the request. The VSE Connector Server then processes the action returned by the *ExecuteHandler* function, which indicates that the server plugin has finished all processing (all actions are completed).

8 The VSE Connector Server calls the *CleanupHandler* function to allow the server plugin to cleanup all resources allocated for this specific request. If an unexpected error occurs during request processing (for example a network connection is broken), the *CleanupHandler* function should be able to cleanup all resources that were used up to this point of time.

9 The VSE Connector Server is shut down, and it calls the *CleanupPlugin* function of all the loaded server plugins.

Implementing a PluginMainEntryPoint Function

The VSE Connector Server calls this function to establish the entry point for a plugin. The *PluginMainEntryPoint* function fetches all other entry points of the plugin's functions. The VSE Connector Server calls *PluginMainEntryPoint* several times. Each call then returns the entry point of the plugin function specified using the *iFuncID* parameter.

Notes:

1. You should not rename the function *PluginMainEntryPoint*. This is because the C-Header file IESPLGIN.H contains a `#pragma linkage` statement which defines this function as `fetchable`.
2. If you do decide to rename this function, you must provide your *own* `#pragama linkage` statement:

   ```
   #pragma linkage(MyEntryPointFunction,fetchable)
   ```

   which forces *MyEntryPointFunction* to be used as entry point of your PHASE.

Here is an example of how the *PluginMainEntryPoint* function can be coded:

```
/*********************************************************************
* Function:    PluginMainEntryPoint Main entry point of the Plugin     *
*                           PHASE.                                    *
* Parameters:  iFuncID      Function ID to get the entrypoint for     *
* Return:      Entrypoint of the requested Function or NULL           *
*********************************************************************/
FUNC_PTR PluginMainEntryPoint(int iFuncID)
{
  switch(iFuncID)
  {
    case PLGFUNC_SETUPPLUGIN:
      return((FUNC_PTR)fetchep((FETCH_PTR)SetupPlugin));
      break;
    case PLGFUNC_CLENUPPLUGIN:
      return((FUNC_PTR)fetchep((FETCH_PTR)CleanupPlugin));
      break;
    case PLGFUNC_GETHANDLEDCMDS:
      return((FUNC_PTR)fetchep((FETCH_PTR)GetHandledCommands));
      break;
    case PLGFUNC_SETUPHANLDER:
      return((FUNC_PTR)fetchep((FETCH_PTR)SetupHandler));
      break;
    case PLGFUNC_EXECUTEHANLDER:
      return((FUNC_PTR)fetchep((FETCH_PTR)ExecuteHandler));
      break;
    case PLGFUNC_CLEANUPHANLDER:
      return((FUNC_PTR)fetchep((FETCH_PTR)CleanupHandler));
      break;

    default:
      return(NULL);
      break;
  };
  return(NULL);
};
```

Figure 143. Sample Code for Implementing PluginMainEntryPoint Function

Implementing a SetupPlugin Function

The VSE Connector Server calls this function *after* a server plugin has been loaded. The *SetupPlugin* function:

- Performs all the initialization steps required for the plugin.
- Gets a pointer to a PLUGIN_INFO block, which contains information about:
  - The z/VSE system parameters that are specified in the plugin configuration member.
  - Entry points of utility functions that the VSE Connector Server provides.

You can also allocate your own plugin private-control block, and pass the pointer of this block back to the VSE Connector Server. The VSE Connector Server passes this pointer to each function belonging to your plugin, that is called from this point-of-time onwards.

Notes:

1. If you define the same plugin more than once, the plugin will be loaded more than once.
2. The *SetupPlugin* is called separately for each plugin “instance”.
3. All occurrences of plugin private-control blocks that you allocate, are separated from each other.

Here is an example of how the *SetupPlugin* function can be coded:

```
/*********************************************************************
* Function:    SetupPlugin   Sets up the Plugin. The Plugin may alloc  *
*                            resources used for command processing.    *
*                            The Plugin can allocate a PluginPrivate   *
*                            Data Area, that is passed to each function*
* Parameters:  lplpPluginPrivate Pointer to a Pointer to the Plugins   *
*                            private Data area. This pointer should be *
*                            set by the Plugin.                        *
*              lpPluginInfo  Pointer to a struct PLUGIN_INFO containing*
*                            several information about the server      *
* Return:      Error values see PLGERR_xxx values                      *
*********************************************************************/
int SetupPlugin(void** lplpPluginPrivate,PLUGIN_INFO* lpPluginInfo)
{
  SAMPLE_PLUGIN*  lpSamplePlugin;

  /* Parameter checking ... */
  if(lplpPluginPrivate==NULL || lpPluginInfo==NULL)
    return(PLGERR_INVALID_PARAM);

  /* Allocate Plugin-Private Data area */
  *lplpPluginPrivate = malloc(sizeof(SAMPLE_PLUGIN));
  if(*lplpPluginPrivate==NULL)
    return(PLGERR_NULL_POINTER);

  /* Fill the Plugin-Info */
  lpPluginInfo->dwPluginVersion = PLUGIN_VERSION;
  strcpy(lpPluginInfo->szDescription,PLUGIN_DESCRIPTION);

  /* Initialize the Plugin ... */

  lpSamplePlugin = (SAMPLE_PLUGIN*)*lplpPluginPrivate;
  /* Open SYSLOG as trace output */
  lpSamplePlugin->lpOutput = fopen("DD:SYSLOG","w");
  if(lpSamplePlugin->lpOutput==NULL)
    return(PLGERR_NULL_POINTER);

  fprintf(lpSamplePlugin->lpOutput,"SetupPlugin Called\n");

  return(PLGERR_NO_ERROR);
};
```

Figure 144. Sample Code for Implementing the SetupPlugin Function

Implementing a CleanupPlugin Function

The VSE Connector Server calls this function *before* a server plugin has been unloaded. It is the counterpart to the *SetupPlugin* function.

The *SetupPlugin* function performs all required cleanup steps. The plugin now de-allocates plugin private-control blocks that were previously allocated.

Here is an example of how the *CleanupPlugin* function can be coded:

```
/*********************************************************************
* Function:    CleanupPlugin Cleans up the Plugin.                  *
* Parameters:  lpPluginPrivate Pointer to the Plugins private data area*
* Return:      Error values see PLGERR_xxx values                   *
*********************************************************************/
int CleanupPlugin(void* lpPluginPrivate)
{
  SAMPLE_PLUGIN*  lpSamplePlugin;

  /* Parameter checking... */
  if(lpPluginPrivate==NULL)
    return(PLGERR_INVALID_PARAM);

  /* Cleanup the Plugin ... */

  lpSamplePlugin = (SAMPLE_PLUGIN*)lpPluginPrivate;

  fprintf(lpSamplePlugin->lpOutput,"CleanupPlugin Called\n");

  /* Close trace output */
  fclose(lpSamplePlugin->lpOutput);

  /* Free the Plugin Private Data */
  free(lpPluginPrivate);

  return(PLGERR_NO_ERROR);
};
```

Figure 145. Sample Code for Implementing the CleanupPlugin Function

Implementing a GetHandledCommands Function

The VSE Connector Server calls this function immediately after *SetupPlugin* has completed. It is used to determine which commands (requests), and how many commands (requests), are currently being handled by a server plugin.

GetHandledCommands passes an array of command-Ids back to the VSE Connector Server.

Here is an example of how the *GetHandledCommands* function can be coded:

```
/*********************************************************************
* Function:    GetHandledCommands Returns a list of CommandIDs that are*
*                             handled by this Plugin.                 *
* Parameters:  lpPluginPrivate Pointer to the Plugins private data area*
*              lpiNumCommands Pointer to a int that should be set by   *
*                             Plugin to the number of Commands         *
*              lpCommandIDs  Pointer to a array of ints. Each element  *
*                             defines one CommandID                    *
* Return:      Error values see PLGERR_xxx values                      *
*********************************************************************/
int GetHandledCommands(void* lpPluginPrivate,
                       int* lpiNumCommands,
                       int** lpCommandIDs)
{
  SAMPLE_PLUGIN*  lpSamplePlugin;

  /* Parameter Checking ... */
  if(lpPluginPrivate==NULL ||
     lpiNumCommands==NULL ||
     lpCommandIDs==NULL)
    return(PLGERR_INVALID_PARAM);

  lpSamplePlugin = (SAMPLE_PLUGIN*)lpPluginPrivate;

  fprintf(lpSamplePlugin->lpOutput,"GetHandledCommands Called\n");

  /* There are 2 Command sHandled by the Sample */
  *lpiNumCommands = 2;

  /* Return a Pointer to the List of Command-IDs */
  *lpCommandIDs = &HandledCommandIDs[0];

  return(PLGERR_NO_ERROR);
};
```

Figure 146. Sample Code for Implementing the GetHandledCommands Function

The `HandledCommandIDs` array is defined as follows:

```
int   HandledCommandIDs[2] = { CMD_SAMPLE_TIME,
                               CMD_SAMPLE_DATE };
```

Implementing a SetupHandler Function

The VSE Connector Server calls this function each time a request is received that must be handled by a server plugin:

1. This function initializes the request handler for the a request.
2. The VSE Connector Server passes a pointer to a CMD_INFO control block to this function.
3. The CMD_INFO control block contains information about the command to handle (for example the command ID of the request). You can allocate a handler private-control block and pass a pointer to it back to the server. The server engine will pass this pointer to each function call belonging to the same request.

Note: It is possible that the same request is executed multiple times. For each 'instance' of request the function *SetupHandler* is called separately. That is, the handler private-control blocks you allocate are separated from each other (you must account for this in your coding).

You must also ensure that your handler is implemented as *reentrant*.

Here is an example of how the *SetupHandler* function can be coded:

```
/*********************************************************************
* Function:    SetupHandler  Sets up a Command Handler.              *
* Parameters:  lpPluginPrivate Pointer to the Plugins private data area*
*              lplpCommandPrivate Pointer to a Pointer to the Handlers *
*                            private data area                       *
*              lpCmdInfo     pointer to a struct CMD_INFO defining the *
*                            actual command                          *
* Return:      Error values see PLGERR_xxx values                    *
*********************************************************************/
int SetupHandler(void* lpPluginPrivate,
                 void** lplpCommandPrivate,
                 CMD_INFO* lpCmdInfo)
{
  SAMPLE_PLUGIN*  lpSamplePlugin;

  /* parameter Checking ... */
  if(lpPluginPrivate==NULL ||
     lplpCommandPrivate==NULL ||
     lpCmdInfo==NULL)
    return(PLGERR_INVALID_PARAM);

  lpSamplePlugin = (SAMPLE_PLUGIN*)lpPluginPrivate;

  /* Initialize the Handler... */
  fprintf(lpSamplePlugin->lpOutput,"SetupHandler Called\n");

  /* check which Command we should handle */
  switch(lpCmdInfo->Command.dwCommand)
  {
    case CMD_SAMPLE_TIME:  /* We are Handling Time-Command */
      /* allocate Command Private datat area */
      ...
      break;

    case CMD_SAMPLE_DATE:   /* We are Handling Date Command */
      /* allocate Command Private datat area */
      ...
      break;
  };
  return(PLGERR_NO_ERROR);
};
```

Figure 147. Sample Code for Implementing the SetupHandler Function

Implementing an ExecuteHandler Function

The VSE Connector Server calls this function to execute the processing of the handler. The VSE Connector Server passes the plugin's private-control block and the handler's private-control block to this function. A pointer is also passed to the CMD_INFO block, which contains information about the command to handle. This block is also used to tell the VSE Connector Server what action it should take after the *ExecuteHandler* function has returned control to it.

The VSE Connector Server runs in a single task only. Therefore, your server plugin is also executed in the one (main) task. As a result, you should ensure that the *ExecuteHandler* function (and all other functions) return control to the VSE Connector Server as quickly as possible. To ensure this, you should split the request processing into several small chunks, where each chunk is executed in a minimum amount of time. For an example of how to split the request processing into several chunks, refer to the “Date and Time Sample plugin” (described in “Using the IBM-Supplied Server Plugin Example” on page 285).

Your request handler can set up a action command in the CMD_INFO block and return control to the VSE Connector Server, if it must wait:
- for an ECB (Event Control Block)
- for a timer
- to receive data from the client
- to send data to the client

The VSE Connector Server will wait for the event you specified and will pass control to your plugin, and will pass control to your plugin, after the event has occurred. Therefore, your plugin should not itself wait, since this would block all parallel-executed tasks being performed by the VSE Connector Server.

You also can set up an action command if you want to send or receive data over the network. Your plugin does not have access to network services directly. Instead, the VSE Connector Server handles the sending or receiving of data over the network, on behalf of your plugin.

Here is an example of how the *ExecuteHandler* function can be coded:

```
...
/* Check which Command to handle */
switch(lpCmdInfo->Command.dwCommand)
{
  case CMD_SAMPLE_TIME:  /* Time Command */
    ...
    break;

  case CMD_SAMPLE_DATE:  /* Date Command */
    ...
    break;

  ...
};
```

Figure 148. Sample Code for Implementing the ExecuteHandler Function

The code fragment of Figure 149 on page 281 shows how to distinguish between multiple requests within one plugin. To split the request processing into several small chunks, the CMD_INFO block contains a field *iState* which can be used to store the actual state of the handler:

```
/* Check in which state we are (0 for first call) */
switch(lpCmdInfo->iState)
{
  case 0:  /* Initial State */
    /* Verify that the Command is correctly sent */
    if(lpCmdInfo->Command.dwDataSize!=0)
      return(PLGERR_PROTOCOL_ERROR);

    /* OK, start processing */
    ...
    /* return from handler without any special action */
     lpCmdInfo->iAction = PLGACT_NOTHING;
     lpCmdInfo->iState = 1;
     break;

   case 1: /* Send the Rresponse */
     /* Send the Response Command */
     /* Setup the COMMAND-Struct for Response */
     memset(&lpCmdInfo->Response,0,sizeof(COMMAND));
     lpCmdInfo->Response.dwCommand = RES_SAMPLE_TIME;
     lpCmdInfo->Response.dwDataSize = sizeof(SAMPLE_TIME);

     /* Setup Action and State */
     lpCmdInfo->iAction = PLGACT_SEND_RESP;
     lpCmdInfo->iState = 2; /* State after Send-Command */
     break;

   case 2: /* state after send reponse */
     ...

}
```

Figure 149. Sample Code for Distinguishing Between Multiple Requests Within a Plugin

When a request has been received, the VSE Connector Server sets the field *iState* to zero. The *ExecuteHandler* function can set this field to a value representing its actual state. The *iState* field will be passed to the *ExecuteHandler* function unchanged at the next call. The VSE Connector Server continues to call the *ExecuteHandler* function each time the client gets dispatched, until the action PLGACT_FINISH is returned.

Implementing a CleanupHandler Function

The VSE Connector Server calls this function after a request has been executed. It is the counterpart of the *SetupHandler* function. This function should cleanup the handler and may free the handler private-control block (if allocated during the *SetupHandler* function).

The *SetupHandler* function is normally called after a request processing has been completed (that is, the *ExecuteHandler* function has set the action to PLGACT_FINISH). If a networking error occurs (for example the connection is broken), the VSE Connector Server calls the *CleanupHandler* function. The *CleanupHandler* function should always be able to cleanup the handler. It should check the handler's state, and cleanup all resources that have been allocated by the handler.

Creating Your Own Plugin Callback Functions

The plugin callback functions described previously are those that your server plugin must implement, and which are called by the VSE Connector Server. However, you may decide to split your code into additional plugin callback functions. You also can decide to split the code into several modules.

As previously stated, a VSE server plugin is designed to be implemented in LE/VSE-C. If you do not want to implement it in C, you may either:

- Implement the complete plugin in a different programming language (such as PL1 or assembler). However, you must ensure that your code is LE/VSE compliant, and supports the calling conventions used by LE/VSE.
- Use the skeleton as a small stub code, which calls functions implemented in a different programming language. This is the recommended method.

Figure 150 is a chunk of code that illustrates the second method (using the skeleton as a small stub code). In this example, a function called *MyASMExecuteFunction* is called (via the symbol MYASMEXE). The calling convention's Operating System passes the required parameters.

```
/* define the ASM function */
#pragma map(MyASMExecuteFunction,"MYASMEXE")
#pragma linkage(MyASMExecuteFunction,OS)

int ExecuteHandler(void* lpPluginPrivate,
                   void* lpCommandPrivate,
                   CMD_INFO* lpCmdInfo)
{
  int rc;

  rc = MyASMExecuteFunction(lpCmdInfo);
  /* TODO errro checking ... */

  return(PLGERR_NO_ERROR);
};
```

Figure 150. Example of Calling a Stub Code Written in a Language Other Than C

Action Codes Supported by the VSE Connector Server

On return from the *ExecuteHandler* function you have to set up the action code in the field *iAction* of the CMD_INFO block. This tells the VSE Connector Server what to do in the next iteration. Possible action codes are:

PLGACT_NOTHING
No special action, the handler requests to be called again as soon as possible.

PLGACT_SEND
The handler wants to send data to the client. The handler has to set up the following fields in the CMD_INFO block:

lpData
Pointer to buffer

dwDataSize
Size of data to send in bytes

Note: the buffer must be located in the handler's private-control block. It can not be a local variable, because a local variable is only valid within the function. The handler will get control after all data has been sent.

PLGACT_RECEIVE
The handler wants to receive data from the client. The handler has to set up the following fields in the CMD_INFO block:

lpData
Pointer to buffer

dwDataSize
Size of data to receive in bytes

Note: the buffer must be located in the handler's private-control block. It can not be a local variable, because a local variable is only valid within the function. The handler gets control after all data has been received.

PLGACT_SEND_RESP
The handler wants to send the response header to the client. The handler has to set up the field Response in the CMD_INFO block. The handler gets control after the response header has been sent.

PLGACT_FINISH
The handler tells the VSE Connector Server that the request processing is completed. The VSE Connector Server calls the *CleanupHandler* function as a result of this action, and will never call the ExecuteHandler function for this request.

PLGACT_CHECKCANCEL
This action is like the PLGACT_NOTHING, but the VSE Connector Server will check if the cancel byte is available to receive. A request is cancelable if the flag FLG_CANCANCEL is specified in the command header. The VSE Connector Server will set the field bWasCanceled to TRUE if the cancel byte has been received.

PLGACT_WAIT
The handler wants to wait for a ECB (Event Control Block). The handler has to set up the field *lpECB* to a pointer to the ECB. The handler gets control after the ECB has been posted.

PLGACT_WAITRECV
This action is like the PLGACT_WAIT but it also waits for the availability of data to receive. That is the handler gets control after the ECB has been posted or any data is available to receive. Note: No data is received in this action.

PLGACT_WAITTIMER
The handler wants to wait for a specified amount of time. The handler has to set up the field *iTimer* in the CMD_INFO block to the number of seconds to wait. The handler gets control after the specified amount of time has expired.

Utility Functions Supported by the VSE Connector Server

The VSE Connector Server supports several *utility functions* that can be called from within a server plugin. The plugin:

- Obtains the entry points of the functions in the PLUGIN_INFO control block, which is passed to the *SetupPlugin* function.
- Stores the entry points of the functions it needs in its plugin private-control block.

These are the utility functions that are supported:

StrToAscii
: Converts an EBCDIC string to ASCII using the code pages configured in the VSE Connector Server configuration members

StrToEbcdic
: Converts an ASCII string to EBCDIC using the code pages configured in the VSE Connector Server configuration members

StrToUpcase
: Converts a string to its uppercase equivalent

StrTrim
: Removes heading and trailing blanks from a string

StrReplace
: Replaces any occurrence of a character in a string with an other character

MatchWildCards
: Checks if a string matches a wildcard filter (with '*' and '?')

EncryptData
: Encrypts a buffer of data using the random XOR algorithm

Decryptdata
: Decrypts a buffer of data using the random XOR algorithm

CheckLibrSecurity
: Checks if access to a Libr resource is guaranteed

CheckPOWERSecurity
: Checks if access to a POWER resource is guaranteed

CheckVSAMSecurity
: Checks if access to a VSAM resource is guaranteed

CheckConsoleSecurity
: Checks if access to a Console resource is guaranteed

CheckICCFSecurity
: Checks if access to a ICCF resource is guaranteed

Using the IBM-Supplied Server Plugin Example

IESPLGIN.H and IESPLGSK.C are members supplied with the VSE Connector Client, that were used for writing the *server plugin* of the “Date and Time plugin” example. When writing your own server plugins, you can also use the contents of these members:

- IESPLGIN.H is a C-header file used for defining the programming interface of a plugin.
- IESPLGSK.C is a skeleton C-program which is the basis of a plugin. This skeleton contains all the functions you require for writing your own server plugins. It also contains comments which provide you with guidance when writing your own plugins.

Registering and Compiling Your Server Plugin

1. You must register your server plugin in the VSE Connector Server's plugin configuration member (normally IESPLGIN.L). This member consist of one or more textual lines, where each line defines one plugin. The syntax of each line is as follows:

   ```
   PLUGIN=<phase name>,PARM=<any kind of parameters>
   ```

   where you use:
   - Keyword PLUGIN to enter the name of your server plugin.
   - Keyword PARM to pass a parameter list to your server plugin. You can use this parameter list to configure your server plugin. **Note:** You must use the keyword PARM even if you do not pass any parameters to the plugin.
2. You must enter your *phase name* in the LIBDEF chain of the VSE Connector Server startup job (see “Job SKVCSSTJ – Startup Job” on page 28 for details).
3. To compile your server plugin, you should use a job like the one below. Your plugins must be compiled using the re-entrant option (RENT).

   ```
   * $$ JOB JNM=COMPILE,CLASS=4,DISP=L
   * $$ LST DISP=D,CLASS=A,RBS=100
   // JOB COMPILE AND LINK VSE CONNECTOR SEVRER PLUGIN
   // LIBDEF *,SEARCH=(YOURLIB.YOURSLIB,PRD2.SCEEBASE,PRD2.PROD,         X
                  PRD2.DBASE)
   // LIBDEF PHASE,CATALOG=YOURLIB.YOURSLIB
   // OPTION ERRS,SXREF,SYM,NODECK,CATAL,LISTX
      PHASE YOURNAME,*,SVA
   // EXEC EDCCOMP,SIZE=EDCCOMP,PARM='LONGNAME RENT SS SOURCE            X
                  INFILE(DD:YOURLIB.YOURSLIB(YOURNAME.C))'
   /*
   // EXEC EDCPRLK,SIZE=EDCPRLK,PARM='UPCASE MAP'
   /*
   // EXEC LNKEDT,SIZE=512K
   /&
   * $$ EOJ
   ```
4. Now you can restart the VSE Connector Server, which will write this message to the system console:

   ```
   Fn nnnn LOADING PLUGIN:  <your plugin>
   ```

Implementing a Client Plugin

For each *server plugin* that you implement (described in "Implementing a Server Plugin" on page 269), you must also implement a *client plugin*. The client plugin consists of one JavaBean for each request that your server plugin supports.

These are the requirements for developing your client plugin. You must:
- Have a JDK (Java Development Kit) installed (Version 1.1.6 or later)
- Have the VSE Connector Client installed. As a minimum, the JAR file **VSEConnector.jar** must be entered in your classpath.
- Use the JavaDoc of the VSE Java Beans as a API reference.

The JavaBeans you implement use the communication services provided by the VSE Java Beans. Therefore, your JavaBeans do not have to establish a connection to the VSE Connector Server, since each of the VSE Java Beans that your plugin uses is a subclass of the *VSEResource* class. Your JavaBeans use an instance of the class *VSESystem* to identify the z/VSE host.

The setup of the *VSESystem* is the same as for all other VSE Java Beans (for example a *VSELibrarian* Bean). That is, you:
1. Create a *VSEConnectionSpec* and set the necessary properties.
2. Create a *VSESystem* and use the *VSEConnectionSpec* to identify the target z/VSE host.
3. Create an instance of your Plugin Bean (when you have a *VSESystem* object).

Here is the outline code for setting up the VSESystem:

```
VSEConnectionSpec spec =  new VSEConnectionSpec(...);
...
VSESystem system = new VSESystem(spec);
...
MyPluginBean myPlugin = new MyPluginBean(system);
```

Using the VSEPlugin class

Any client plugin JavaBean that you write must be a subclass of *VSEPlugin*. *VSEPlugin* is a abstract class, and is itself a subclass of *VSEResource*. As you can also see in the JavaDoc of the VSE Java Beans, *VSEResource* implements the following methods:

setVSESystem
Sets the *VSESystem* to use

getVSESystem
Gets the currently used *VSESystem*

addVSEResourceListener
Adds a *VSEResourceListener* to the bean

removeVSEREsourceListener
Removes a *VSEResourceListener*

notifyListStarted
Notifies all registered *VSEResourceListeners*

notifyListAdded
Notifies all registered *VSEResourceListeners*

notifyListEnded
Notifies all registered *VSEResourceListeners*

setCancel
Cancels the current request

getCancel
Gets the current cancel state

isExistent
Checks if the resource is existent

toString
Returns a string representation

In addition you may also decide to implement some of your own methods, to add properties to your JavaBean.

To activate the request of the client plugin JavaBean, call the method *execute()*. This method is declared as protected, since it should only be called from inside the class. For example you may override the method *refresh* and call *execute* to request a refresh of the JavaBeans properties. The method *execute* will trigger the execution of the request and will return control, after the request has been processed. During request processing, these methods are called:

- *getRequestID*
- *getRequestSubcommand*
- *getRequestFlags*
- *getRequestError*
- *getRequestDataSize*
- *sendRequestdata*
- *getResponseID*
- *getResponseDataSize*
- *receiveResponseData*

For the server plugin and client plugin to be able to work together:

- The Command Ids returned by the methods *getRequestID* and *getResponseID* must match the CommandIDs used in the server plugin.
- The data format that is transferred for a request or response, must match the data format that is supported by the server plugin.
- The plugin JavaBean must use the protocol used for communicating between the plugin server and the plugin client.

The methods *getRequestDataSize* and *getResponseDataSize* should return the size (in bytes) of the data that is transferred:

- If no data is transferred, this method should return zero.
- If the request or response use a DATAEX format, the method should return the constant SIZE_UNKNOWN. This flags the command as a DATAEX command.
- The methods *sendRequestData* and *receiveResponseData* should also return the size of the data that has been sent or received.

In addition:

- For a DATAEX command, the method should retain control until the command has finished. After completion of the data transfer the method should return control to indicate the end of the command.

- For a non-DATAEX command, the return value of the *sendRequestData* and *receiveResponseData* methods should be equal to the value returned by *getRequestDataSize* or *getResponseDataSize*.

Your client plugin can call these methods:
- *getResponseSubCommand*
- *getResponseFlags*
- *getResponseError*

in the method *receiveResponseData()* or after return of control from method *execute()*. By calling these methods, your client plugin can obtain the sub command, flags and error values that were sent in the response.

Your client plugin can then use these values to perform error checking and handling.

General Considerations When Designing Your Plugin

You should consider these areas when designing your plugin:
- How should I define the protocol between the VSE Connector Server and my server plugin?.
- How can I access the resource or application on z/VSE?.
- What kind of data should be accessed?.
- Which requests or functions should be allowed?.
- How is the data format to be transferred over the network?.
- How should I structure the view of the data for the client plugin, when defining the JavaBean interface?.

Specifying the Protocol Between VSE Connector Server and Plugin

The Java-based connector uses its *own protocol* for communication purposes. However, you can extend this protocol with your own commands. The characteristics of the protocol used by the Java-based connector is described in this topic.

The protocol used by the Java-based connector is based on a TCP/IP connection: it uses a *connection-orientated* stream to transfer data. The (endless) stream is separated into smaller parts, called *stream commands*. A stream command consist of:
- a command header
- a data part (optional)

The various stream commands are identified by a unique 4-byte command-Id, which is stored in the command header. The command header can contain these optional fields:

Field **Description**

Flags Bit-flags (4 bytes). For details, see predefined Flag values

Error Error code (4 bytes). For details, see predefined error codes

SubCommand
SubCommand ID (4 bytes). For future use only.

The command header is directly followed by the data belonging to the stream command. If the stream command contains no data, the command header for the next stream command follows.

The data belonging to a stream command can be transferred in two ways:
- Fixed-length data
- Variable-length data (DATAEX)

When data is transferred as *fixed length*, the number of data bytes is defined in the command header. After this number of bytes, the next command header starts.

When data is transferred as *variable length* (the “DATAEX” method), the length of the data is not directly specified, but is instead specified implicitly within the data. When developing your plugin, you can therefore define any kind of indicator within the transferred data, that marks the end of the data. The sender and the receiver of the data must both be able to recognize this indicator. After the end of the variable length data has been reached, the next command header follows.

When developing your plugins, you can extend the existing set of requests with your own requests. To do so, you must define:
- Command Ids for your requests and responses.
- The format of the data that belongs to the commands.

Choosing the Access Method to the Data / Application

Your must first choose the access method on z/VSE to be used for accessing the data or application for which you require a plugin. Since the VSE Connector Server runs in a *batch* partition (static or dynamic), your plugin will also be executed in the same batch environment. As a result, your access method must be able to access the data or application from within a *batch* program.

Your plugin must be written within the LE/VSE environment. A VSE Connector Server plugin should be implemented in LE/VSE C, since the VSE Connector Server is itself implemented in LE/VSE C. You can, however, call Assembler or PL/1 modules from within your plugin.

You also can use VSE macros (such as GETVIS) when writing your plugin, but you must program the use of the LE/VSE register yourself.

Considerations for ASCII / EBCDIC and Big / Little Endian

You can provide access to any kind of data. If you decide to access textual data, you must use the utility functions to convert between ASCII and EBCDIC, before transferring the data over the network. Your plugin is responsible for carrying out the required conversions.

You can perform the conversion between ASCII and EBCDIC on the client side, but you are recommended to perform this conversion on the server side. The utility function to convert form ASCII to EBCDIC and vice versa uses the configured code pages to do the translation. If you decide to transfer binary data like integer values, you do not usually have to be concerned about big or little endian formats.

z/VSE (that is the System z platform) uses the big endian format for integer values. Java also uses the big endian format even if it is running on an Intel (little endian) platform. Therefore you do not have to convert integer values before transferring them.

Deciding Which Requests / Functions Should Be Supported

You have to decide which kind of access requests or functions your plugin should support. A VSE Connector Server plugin can support one ore more different requests.

You can decide to use one request for each kind of access (read, write, update, ...), or you can decide to use only one request which is able to execute different kinds of access. In the latter case, you might define a parameter which informs the plugin when a write or read operation is requested.

Transferring Data Over the Network

You must also define the interface between the server-part and the client-part of the plugin. That is, you must define the *protocol* that is used to transfer the data. A well-defined protocol is used for communicating between the VSE Connector Client and VSE Connector Server. Therefore, your plugin's protocol must also support this protocol.

The protocol used for communicating between the VSE Connector Client and VSE Connector Server consists of *requests*. Each request is:

- Independent from other requests.
- Transferred as a block over the network.
- Identified by a 4-byte number, the *command-Id*.

Your plugin can use its own command-Ids that must be within a specific range. Although your plugin must use the predefined protocol to transfer its data, the format of the data itself can be defined by yourself.

Structuring the Client Plugin's View

You must also consider how the client-part view of the data or application, should be structured. The Java-based connector allows you to implement your own JavaBeans plugins which use the communication methods provided by the Java-based connector.

For each request you have to implement a corresponding JavaBean on the client side. This JavaBean is responsible for sending and receiving the data belonging to the request. Therefore, this JavaBean must be familiar with the format of the transferred data.

You can design the external interfaces of the plugin JavaBeans that you develop.

Chapter 22. Using the Database Call Level Interface to Access Data

This chapter describes how your application programs can use the *Database Call Level Interface* (DBCLI) to access a relational database on any suitable database server. The database server usually runs on a non-z/VSE platform.

This chapter contains these main topics:

- “DBCLI Programming Concepts”
- “Programming Restrictions and Requirements” on page 296
- “Using the DBCLI in COBOL” on page 296
- “Using The DBCLI in PL/I” on page 297
- “Using The DBCLI in C” on page 297
- “Using the DBCLI in Assembler” on page 298
- “Using the DBCLI in REXX” on page 299
- “Syntax and Parameters of a DBCLI Function Call” on page 299
- “DBCLI Functions (Reference Information)” on page 300
- “Performance Considerations When Using the DBCLI” on page 390
- “Investigating the Cause of Errors When Using the DBCLI” on page 391
- “Return Codes Used by the DBCLI” on page 392

Related Topics:

- Chapter 9, “Installing the Database Call Level Interface,” on page 73 (which includes an *overview* of the Database Call Level Interface).
- COBOL example **COBSAMPL** which is shipped as part of the DBCLI server package. This example illustrates how to use the Database Call Level Interface within a COBOL program.

DBCLI Programming Concepts

Initializing and Terminating the API Environment

When using the API provided by the DBCLI client, you must:

1. Initialize the API environment by calling the `INITENV` function before calling any other function. The `INITENV` function allocates an *environment handle* that you must pass to all subsequent functions. You can have *only one active environment* at a time in your program.
2. Terminate the API environment (at the end of your program) by calling the `TERMENV` function. The `TERMENV` function frees all resources allocated by the DBCLI code. The `TERMENV` function will also close any “left over” connections or statements. After the `TERMENV` function, the environment handle is *no longer valid*.

You can set and get various attributes on the environment level. You do so by calling the `SETENVATTR` or `GETENVATTR` functions.

© Copyright IBM Corp. 2000, 2013

Connecting and Disconnecting to/from the DBCLI Server and Vendor Database

Before you can use a particular database, you must connect to the *DBCLI server* (DBCliServer) and the Vendor database.

- You connect to the DBCLI server (DBCliServer) and the Vendor database by calling the CONNECT function. You must supply the:
  - IP address or hostname of DBCliServer.
  - Alias name of the database to which you wish to connect.
- The CONNECT function allocates a *connection handle* that you must pass to all subsequent functions that require a connection.
- You can have *multiple connections* to the same or different DBCLI servers and databases at any one time.
- Each connection is represented by its own connection handle. It is the applications responsibility to pass the required connection handle to subsequent functions.

When you are finished working with a database, you must disconnect from the database and the DBCLI server (DBCliServer) by calling the DISCONNECT function.

- The DISCONNECT function frees the connection handle and all left over statements (if any) that you have allocated using this connection.
- The connection handle is no longer valid after the DISCONNECT function.

You can set and get several attributes on the connection level. You do so by calling the SETCONNATTR or GETCONNATTR functions.

Any SQL statements that you prepare are strongly bound to the connection they were created with. You can not use a statement created on one connection with another connection. You will get an error if you try to do so.

Logical Units of Work (Transactions)

Per default, a connection operates in *transaction mode*:

- Any database updates that you perform using the connection are contained in a *logical unit of work*.
- You can end a logical unit of work by calling the COMMIT or ROLLBACK functions.
  - The COMMIT function commits all changes done since the beginning of the logical unit of work and starts a new logical unit of work.
  - The ROLLBACK function rolls back (reverts) all changes since the beginning of the logical unit of work or up to a savepoint.

Usually, you should explicitly call the COMMIT function at the end of the program.

- If you do not call the COMMIT function, DBCliServer will automatically commit all changes, if you gracefully close the connection by calling the DISCONNECT function.
- If the connection is dropped (for example, because the program abends), the DBCLI server (DBCliServer) rolls back all changes done since the beginning of the last logical unit of work.

You can set a connection into *auto-commit mode*.

- In auto-commit mode, every SQL statement is treated as its own logical unit of work and is committed automatically when the statement execution is complete. Therefore, you do not have to call the COMMIT or ROLLBACK functions.

- You set a connection into auto-commit mode by calling the SETCONNATTR function to set the CONNATTR-AUTO-COMMIT attribute to TRUE.

Executing SQL Statements

In order to execute an SQL statement, you must first *prepare the SQL statement*.

- During preparation, the database will usually precompile the SQL statement and creates an *access plan* for the statement.
- The access plan is kept as long as the statement exists.
- You can then execute the statement as many times as you want.

The PREPARESTATEMENT function prepares an SQL statement for execution. It allocates a *statement handle* that represents the statement.

- The application must pass the statement handle to all subsequent functions that require a statement.
- An application can prepare multiple statements at-a-time.
- The application must ensure it uses the required statement handle with subsequent functions.

The PREPARECALL function prepares a stored procedure call statement for execution.

SQL statements may contain *parameters* that are evaluated at execution time.

- Parameters are marked by a question mark (?) within the SQL statement.
- The parameters are numbered in order of appearance, starting with 1.
- After preparing, the application can bind host variables to the parameters using the BINDPARAMETER function. When the statement is later executed, the content of the host variables is used and sent to the database.

The following example SELECT query contains one parameter:

```
SELECT * FROM EMPLOYEE WHERE SALARY>?
```

You can use the GETNUMPARAMETERS and GETPARAMETERINFO functions to obtain detailed information about the statement parameters.

To execute a statement, you must call the EXECUTE function.

- If the statement was an update SQL statement, you can retrieve the number of rows updated using the GETUPDATECOUNT function or the UPDATE-COUNT parameter at the EXECUTE function.
- If the statement was a SQL query, you can use a cursor to retrieve (fetch) the result rows and columns.
- A statement can provide multiple results.
  - To retrieve the additional results you must call the GETMORERESULTS function.
  - The GETMORERESULTS function will move to the next available cursor or update count.
  - If the statement was a stored procedure call, you must call the GETMORERESULTS function until no more results are available. Only then will the output parameters be updated with the data passed back by the stored procedure.

You can set and get several attributes on the statement level. You do so by calling the SETSTMTATTR or GETSTMTATTR functions.

When you no longer need a statement, you must close it by calling the CLOSESTATEMENT function:

- The CLOSESTATEMENT function frees the statement handle and closes all cursors (if any) that may still be open from the last statement execution.
- The statement handle is no longer valid after the CLOSESTATEMENT function.

Cursors

The execution on an SQL query returns a result in form of a *cursor*.

- A cursor allows you to retrieve (fetch) the result rows and columns.
- You can use the GETNUMCOLUMNS and GETCOLUMNINFO functions to obtain detailed information about the cursor's columns.
- The columns are numbered in order of appearance, starting at 1.

To fetch the result rows using the cursor, you must first *bind* host variables to the columns of interest.

- You bind host variables to the columns of interest by calling the BINDCOLUMN function.
- If the FETCH function is called later on, the host variables will be updated with the contents of the column in the row that has been fetched.
- Per default, the FETCH function processes the cursor from the beginning to the end.

You may *reposition* with a cursor, providing:

- The database supports this.
- You have created the statement using the appropriate type (CURSOR-TYPE-SCROLL-INSENSITIVE or CURSOR-TYPE-SCROLL-SENSITIVE).

Repositioning can be performed using either the:

- FETCH function with operations FETCH-PREVIOUS, FETCH-FIRST, FETCH-LAST, FETCH-ABSOLUTE or FETCH-RELATIVE.
- SETPOS function.

You may also update, delete or insert rows *within* a cursor, providing:

- The database supports this.
- You have created the statement using the appropriate type (CURSOR-TYPE-SCROLL-INSENSITIVE or CURSOR-TYPE-SCROLL-SENSITIVE) and concurrency (CURSOR-CONCUR-UPDATABLE).

The update, delete or insert is performed by calling the SETPOS function using the operations SETPOS-UPDATE, SETPOS-DELETE or SETPOS-INSERT.

When you no longer need a cursor, you must close it by calling the CLOSECURSOR function. You can no longer use the FETCH function to retrieve rows after the CLOSECURSOR function.

Database Meta Data

The DBCLI interface allows you to retrieve *meta data* from the database. This includes functions to get a list of tables, indexes, keys, columns of a table, and so on. This information is typically stored in system catalog tables in the database. You can also execute regular SELECT statements against the system catalog tables, but this requires that you know which database system and vendor you are using. System catalog tables are vendor- and database-specific.

The DBCLI interface provides a set of database independent functions to retrieve meta data information. These functions are prefixed with 'DB'. The function DBTABLES for example retries a list of tables available in the database.

Please note that some databases may not support all of the meta data functions.

Connection Pooling

Connection pooling keeps and reuses *existing* database connections for DBCLI applications running under the CICS Transaction Server for VSE/ESA V1.1. It is implemented by configuring the *connection pool manager*, as described in "Configuring and Starting/Stopping the Connection Pool Manager" on page 78.

The default is to **not** use connection pooling. Therefore, existing DBCLI applications will continue to work unchanged (without using the connection pool).

The connection pool is only available *under CICS*. Batch applications cannot make use of the connection pool. An attempt to use the connection pool while running in batch will be rejected.

Your DBCLI applications can use SSL (Secure Socket Layer) for connections between z/VSE and the DBCLIServer. However:

- Pooling of SSL connections is *not* supported.
- An attempt to use the connection pool with an SSL connection will be rejected.

DBCLI applications can request to use the connection pool by setting environment variable ENVATTR-USE-CONNPOOL to TRUE prior to calling the CONNECT function. In this case:

- No further applications changes are required.
- The use of the connection pool is *transparent* to DBCLI applications.

When connection pooling is enabled by an application:

1. The CONNECT function will check if a *matching* connection is available in the connection pool (that is, a connection with the same hostname/IP address, port, database name, user-ID and password).
   a. If a matching connection *is* available, the matching connection is obtained from the pool and is reused.
   b. If the connection is no longer active or if a matching connection cannot be obtained from the pool, a new connection is established.
2. When the application has finished using the connection, it executes a DISCONNECT function. During DISCONNECT handling, the connection is put back into the connection pool so that it can be reused at a later time.
3. Regardless of whether or not the connection pool is used, a graceful DISCONNECT call will commit the changes to the database.

a. If the application program abends or terminates *without* gracefully disconnecting from the database, the changes are automatically rolled back.
b. In this case, the connection is not put back into the pool but is closed.

Related Topics:
- "Overview of Connection Pooling" on page 74.
- "Configuring and Starting/Stopping the Connection Pool Manager" on page 78.

Programming Restrictions and Requirements

The following restrictions apply to the DBCLI programming interface:

- The DBCLI code is CICS-aware. If running under CICS, any memory allocations are performed using EXEC CICS GETMAIN instead of using the GETVIS macro.
- The DBCLI API cannot be used with programs running in an ICCF Pseudo Partition.
- Locks should *not* be held when issuing these DBCLI calls.
- When using the DBCLI API in CICS transactions while CICS operates with storage protection, all programs using the DBCLI API need to be defined with EXECKEY(CICS). This is also true for those programs that link to these programs. TASKDATAKEY(CICS) for the transaction definition is NOT required.
- When using the DBCLI API in CICS transactions, the EZA "task-related-user-exit" (TRUE) has to be activated before these transactions can be run. For details on how to activate this TRUE, refer to "CICS Considerations for the EZA Interfaces" in the *z/VSE TCP/IP Support*.
- Most JDBC drivers will only accept pure SQL statements. They will *not* accept SQL preprocessor statements that are used for DB2 Server for VSE applications.
- The connection pool is only available under CICS. Batch applications *cannot* use the connection pool. Any attempt to use the connection pool while running in batch will be rejected.
- DBCLI supports the use of SSL (Secure Socket Layer) for the connection between z/VSE and the DBCLIServer. However, the pooling of SSL connections is *not* supported. An attempt to use the connection pool with an SSL connection will be rejected.

Using the DBCLI in COBOL

To use DBCLI within a COBOL program, the following general considerations apply:

```
►►──CALL 'IESDBCLI' USING FUNCTION ENV-HANDLE──parm1 parm2 ... parmN──RETCODE.────►◄
```

All parameters must be passed by reference. This is the default for COBOL.

FUNCTION
A 16-byte character field, left-justified and padded on the right with blanks. Set to the name of the call. FUNCTION is case specific. It must be in uppercase. The function must always be the first parameter.

ENV-HANDLE
A fullword binary variable containing the environment handle. The ENV-HANDLE parameter must be specified with every call to DBCLI as the second parameter.

parmn
A variable number of parameters depending on the type call.

RETCODE
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The COBOL copybook IESDBCOB contains common declarations.

Using The DBCLI in PL/I

To use DBCLI within a PL/I program, the following general considerations apply:

```
►►──CALL IESDBCLI(FUNCTION,ENV_HANDLE,parm1,parm2,...,parmN,RETCODE);──────────►◄
```

All parameters must be passed by reference. This is the default for PL/I.

FUNCTION
A 16-byte character field, left-justified and padded on the right with blanks. Set to the name of the call. FUNCTION is case specific. It must be in uppercase. The function must always be the first parameter.

ENV_HANDLE
A fullword binary variable containing the environment handle. The ENV-HANDLE parameter must be specified with every call to DBCLI as the second parameter.

parmn
A variable number of parameters depending on the type call.

RETCODE
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The PL/I copybook IESDBPL1 contains common declarations. It also includes the following statement:

```
DCL IESDBCLI ENTRY OPTIONS(RETCODE,ASM,INTER) EXT;
```

Using The DBCLI in C

To use DBCLI within a C program, the following general considerations apply:

```
char function[16];
void* env_handle;
int retcode;
```

```
►►──IESDBCLI(function,&env_handle,&parm1,&parm2,...,parmN,&retcode);        ──────►◄
```

All parameters must be passed by reference. That is you must explicitly supply the address of the parameter (use the & operator) to the IESDBCLI function. For C-strings, you do not need to use the & operator, since C-strings are arrays of characters, and are passed by reference anyway.

Note that DBCLI does not support the *C zero-termination*:

- For input parameters, you must supply a C-string of the required length, padded with blanks.

- You may add a zero termination character after that, but it will be ignored by DBCLI.
- For output parameters, DBCLI does not terminate the string using a zero character.
- It is up to you to ensure proper zero-termination, if required.

function
A 16-byte character field, left-justified and padded on the right with blanks. Set to the name of the call. FUNCTION is case specific. It must be in uppercase. The function must always be the first parameter.

env_handle
A fullword binary variable containing the environment handle. The ENV-HANDLE parameter must be specified with every call to DBCLI as the second parameter.

parmn
A variable number of parameters depending on the type call.

retcode
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The C header file IESDBC.h contains common declarations. It also contains the following C function prototype and #pragma statements:

```
int IESDBCLI(char* function, void** env_handle, ...);
#pragma map(IESDBCLI,"IESDBCLI")
#pragma linkage(IESDBCLI,OS)
```

Using the DBCLI in Assembler

To use DBCLI within an Assembler program, the following general considerations apply:

```
▶▶──CALL IESDBCLI,(FUNCTION,ENV_HANDLE,parm1,parm2,...,parmN,RETCODE),VL──────────▶◀
```

You can use the following call format for reentrant programming.

```
▶▶──CALL IESDBCLI,(FUNCTION,ENV_HANDLE,parm1,parm2,...,parmN,RETCODE),VL,MF=(E,list-addr)────────▶◀
```

All parameters must be passed by reference. That is, the parameter list pointed to by register 1 must contain the addresses of the parameters. The last entry of the parameter list must have the high order bit on to indicate the last parameter. The CALL macro takes care about this already when you use the VL option as recommended.

FUNCTION
A 16-byte character field, left-justified and padded on the right with blanks. Set to the name of the call. FUNCTION is case specific. It must be in uppercase. The function must always be the first parameter.

ENV_HANDLE
A fullword binary variable containing the environment handle. The ENV-HANDLE parameter must be specified with every call to DBCLI as the second parameter.

parmn
A variable number of parameters depending on the type call.

RETCODE
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The following register conventions apply:

- Register 0, 1, 14, and 15 are used by the interface and must be, if necessary, saved prior to invocation.
- Register 13 must point to a 72-byte save area provided by the caller.

The Assembler macro IESDBASM contains common declarations.

Using the DBCLI in REXX

To use DBCLI within an REXX program, you must use the ADDRESS LINKPGM host command environment to call external program IESDBCLA:

```
►►──ADDRESS LINKPGM "IESDBCLA FUNCTION ENV_HANDLE parm1 parm2 ... parmN RETCODE"────────►◄
```

All parameters must be initialized with a value of the appropriate length before calling the DBCLI API. This is especially true for output parameters. Fullword binary variables must be initialized to contain 4 bytes (for example, VARIABLE = D2C(0,4)). Since the variable is expected to contain a value in binary representation, you must convert the value from the REXX string representation into the binary representation and vice versa using the REXX functions C2S and D2C.

FUNCTION
A 16-byte character field, left-justified and padded on the right with blanks. Set to the name of the call. FUNCTION is case specific. It must be in uppercase. The function must always be the first parameter.

ENV_HANDLE
A fullword binary variable containing the environment handle. The ENV-HANDLE parameter must be specified with every call to DBCLI as the second parameter.

parmn
A variable number of parameters depending on the type call.

RETCODE
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

Syntax and Parameters of a DBCLI Function Call

The programming interface provided by DBCLI allows your own applications to call the DBCLI client from batch or CICS TS applications written in COBOL, PL/1, C, Assembler or REXX.

The parameter descriptions in this topic are written using the COBOL for VSE language syntax and conventions, but you should use the syntax and conventions that are appropriate for the language you want to use.

These are examples of storage definition statements for COBOL, PL/I, C, and assembler language programs:

```
COBOL PIC
  PIC S9(4) BINARY                                  HALFWORD BINARY VALUE
  PIC S9(8) BINARY                                  FULLWORD BINARY VALUE
  PIC    X(n)                                       CHARACTER FIELD OF N BYTES
PL/I DECLARE STATEMENT
  DCL    HALF        FIXED BIN(15),                 HALFWORD BINARY VALUE
  DCL    FULL        FIXED BIN(31),                 FULLWORD BINARY VALUE
  DCL    CHARACTER   CHAR(n)                        CHARACTER FIELD OF n BYTES
C DECLARATION
  short  half;                                      HALFWORD BINARY VALUE
  int    full;                                      FULLWORD BINARY VALUE
  char   string[n];                                 CHARACTER FIELD OF n BYTES
ASSEMBLER DECLARATION
  DS     H                                          HALFWORD BINARY VALUE
  DS     F                                          FULLWORD BINARY VALUE
  DS     CLn                                        CHARACTER FIELD OF n BYTES
```

Several function use optional parameters. Optional parameters are indicated by square brackets as shown in the following example:

```
►►──CALL 'IESDBCLI' USING ENV-HANDLE FUNCTION──parm1 [optparm2] ... [optparmN]────►

►─RETCODE.──────────────────────────────────────────────────────────────────────►◄
```

DBCLI Functions (Reference Information)

Roadmap of Where the DBCLI Functions Are Used

Table 8 provides a “roadmap” that should help you to locate the functions you require.

Table 8. Roadmap of Where DBCLI Functions Are Used

Category	Functions	
Environment functions	“INITENV” on page 379	“GETENVATTR” on page 369
	“TERMENV” on page 389	“GETLASTERROR” on page 371
	“SETENVATTR” on page 386	“INITSSL” on page 380
Connection functions	“CONNECT” on page 309	“COMMIT” on page 308
	“CONNECTSSL” on page 311	“ROLLBACK” on page 385
	“DISCONNECT” on page 350	“SETSAVEPOINT” on page 388
	“SETCONNATTR” on page 385	“RELEASESAVEPOINT” on page 384
	“GETCONNATTR” on page 355	
Statement functions	“PREPARESTATEMENT” on page 382	“SETSTMTATTR” on page 389
	“PREPARECALL” on page 381	“GETSTMTATTR” on page 376
	“CLOSESTATEMENT” on page 308	
Statement-parameter functions	“GETNUMPARAMETERS” on page 374	“BINDPARAMETER” on page 304
	“GETPARAMETERINFO” on page 374	
Statement-execution functions	“EXECUTE” on page 351	“GETUPDATECOUNT” on page 378

Table 8. Roadmap of Where DBCLI Functions Are Used (continued)

Category	Functions	
Cursor functions	"GETNUMCOLUMNS" on page 373	"FETCH" on page 352
	"GETCOLUMNINFO" on page 353	"SETPOS" on page 387
	"BINDCOLUMN"	"GETMORERESULTS" on page 372
	"GETROWNUMBER" on page 376	"CLOSECURSOR" on page 307
Database meta-data functions	"DBATTRIBUTES" on page 313	"DBPROCEDURES" on page 335
	"DBBESTROWIDENT" on page 315	"DBSCHEMAS" on page 337
	"DBCATALOGS" on page 317	"DBSUPERTABLES" on page 338
	"DBCOLUMNPRIV" on page 318	"DBSUPERTYPES" on page 339
	"DBCOLUMNS" on page 320	"DBTABLEPRIV" on page 341
	"DBCROSSREFERENCE" on page 322	"DBTABLES" on page 343
	"DBEXPORTEDKEYS" on page 325	"DBTABLETYPES" on page 345
	"DBIMPORTEDKEYS" on page 327	"DBTYPEINFO" on page 346
	"DBINDEXINFO" on page 329	"DBUDTS" on page 347
	"DBPRIMARYKEYS" on page 332	"DBVERSIONCOLS" on page 349
	"DBPROCEDURECOLS" on page 333	

BINDCOLUMN

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The BINDCOLUMN function allows an application to bind a host variable to a column contained in the cursor after a statement has been executed. When the FETCH function is called later on, the host variable will be set to the value of the column.

Related functions:
- "GETCOLUMNINFO" on page 353
- "GETNUMCOLUMNS" on page 373

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE            PIC S9(8) BINARY.
   01 STMT-HANDLE           PIC S9(8) BINARY.
   01 COL-INDEX             PIC S9(8) BINARY.
   01 NATIVE-TYPE           PIC S9(8) BINARY.
   01 VAR                   PIC .....
   01 VAR-LEN               PIC S9(8) BINARY.
   01 VAR-IND               PIC S9(8) BINARY.
   01 OPTION                PIC ....
   01 OPT-LEN               PIC S9(8) BINARY.
   01 RETCODE               PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-BINDCOLUMN ENV-HANDLE
        STMT-HANDLE COL-INDEX NATIVE-TYPE VAR VAR-LEN VAR-IND
        [OPTION [OPT-LEN]] RETCODE.
```

FUNCTION (input)
A 16-byte character field containing BINDCOLUMN. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

STMT-HANDLE (input)
A fullword binary variable containing the statement handle.

COL-INDEX (input)
A fullword binary variable containing the number of the column to bind. Column numbers start at 1. Column number 0 is a special column containing the current row number. If use, it must be bound to a fullword binary variable with native type NATIVE-TYPE-INTEGER-UNSIGNED.

NATIVE-TYPE (input)
A fullword binary variable containing the native type of the host variable bound to the column. A native type value of 0 causes the column to be unbound. The following native types are available:

NATIVE-TYPE-STRING (1)
A textual string

NATIVE-TYPE-INTEGER-SIGNED (2)
A signed binary integer.

NATIVE-TYPE-INTEGER-UNSIGNED (3)
An unsigned binary integer.

NATIVE-TYPE-PACKED-SIGNED (4)
A signed packed decimal (COMP-3).

NATIVE-TYPE-PACKED-UNSIGNED (5)
An unsigned packed decimal.

NATIVE-TYPE-ZONED-SIGNED (6)
A signed zoned decimal.

NATIVE-TYPE-ZONED-UNSIGNED (7)
An unsigned zoned decimal.

NATIVE-TYPE-C-STRING (8)
A zero terminated string (C language string).

NATIVE-TYPE-FLOAT_BFP (9)
A floating point number in IEEE binary floating point representation (BFP).

NATIVE-TYPE-FLOAT_HFP (10)
A floating point number in hexadecimal floating point representation (HFP).

NATIVE-TYPE-TOD (11)
A 8 byte time of day clock (TOD) value.

NATIVE-TYPE-BOOLEAN (12)
A boolean value, i.e. a bit that is TRUE (1) or FALSE (0).

NATIVE-TYPE-BINARY (13)
A binary field.

NATIVE-TYPE-DATE_PACKED (32)
A date value in packed decimal format. Use the pattern option to specify the date pattern.

NATIVE-TYPE-TIME_PACKED (33)
A time value in packed decimal format. Use the pattern option to specify the time pattern.

NATIVE-TYPE-TIMESTAMP_PACKED (34)
A date and time value in packed decimal format. Use the pattern option to specify the date and time pattern.

NATIVE-TYPE-DATE_STRING (35)
A date value in text format. Use the pattern option to specify the date pattern.

NATIVE-TYPE-TIME_STRING (36)
A time value in text format. Use the pattern option to specify the time pattern.

NATIVE-TYPE-TIMESTAMP_STRING (37)
A date and time value in text format. Use the pattern option to specify the date and time pattern.

VAR (deferred output/input)
A host variable that is bound to the column. When the `FETCH` function is called later on, the host variable will be set to the value of the column. The parameter VAR-LEN specifies the length of the host variable in bytes. Use the COBOL term 'LENGTH OF' to determine the byte length of a variable.

When using REXX, this parameter must be a variable containing the bind variable name as a 32-character text string, padded with blanks. For this purpose, you can use the REXX function LEFT:

```
LEFT('variable-name',32)
```

VAR-LEN (input)
A fullword binary variable contains the length of the host variable in bytes. Use the COBOL term 'LENGTH OF' to determine the byte length of a variable.

VAR-IND (deferred output/input)
A fullword binary variable that acts as an indicator for the host variable bound to the column. The indicator variable is set when the `FETCH` function is called later on. Its content indicates if the variable value is NULL or not, or if the row has been updated, deleted or inserted. For updateable cursors, the indicator variable is also used to specify which columns are to be updated when using the SETPOS function with operation SETPOS-UPDATE.

When using REXX, this parameter must be a variable containing the bind variable name as a 32-character text string, padded with blanks. For this purpose, you can use the REXX function LEFT:

```
LEFT('variable-name',32)
```

The indicator variable can contain the following values:

INDICATE-NOTNULL (0)
The variable value is not NULL. The content of the host variable bound to the column is set.

INDICATE-NULL (1)
The variable value is NULL.

INDICATE-IGNOREUPDATE (2)
This column should be ignored when performing a row update using the SETPOS function with operation SETPOS-UPDATE.

INDICATE-DELETED (16)
Indicates that the current row was deleted.

INDICATE-INSERTED (32)
Indicates that the current row was inserted. This value can be combined with INDICATE-NULL.

INDICATE-UPDATED (64)
Indicates that the current row was updated. This value can be combined with INDICATE-NULL.

OPTION (input, optional)
A variable that is used to specify an option value for a host variable. This parameter is optional. The following native types accept an option value:

NATIVE-TYPE-STRING and NATIVE-TYPE-C-STRING
The option value is a textual field specifying the EBCDIC code page that is used to convert the host variable content from EBCDIC to ASCII and vice versa. The OPT-LEN parameter must be specified and must contain the length of the textual field.

All numeric types (PACKED, ZONED, INTEGER)
The option value is a fullword binary variable containing the implied decimal position used for the numeric data. The OPT-LEN parameter is not required for this option.

NATIVE-TYPE-BOOLEAN
The option value is a fullword binary variable containing the bit position counted from the right. Default is bit zero (right most bit). The OPT-LEN parameter is not required for this option.

All DATE, TIME and TIMESAMP types
The option value is a textual field specifying the pattern that is used to convert the host variable content into a database DATE, TIME or TIMESTAMP value and vice versa. Sample pattern format: "yyyy.MM.dd hh:mm:ss" will accept input like "1996.07.10 15:08:56". The OPT-LEN parameter must be specified and must contain the length of the textual field.

OPT-LEN (input, optional)
A fullword binary variable containing the length of the OPTION parameter if the OPTION is a textual field. This parameter is optional. If it is specified, the OPTION parameter must also be specified.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

BINDPARAMETER

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The BINDPARAMETER function allows an application to bind a host variable to a statement parameter. When the statement is executed later on, the contents of the host variable bound to the parameter is sent to the database.

Related sections:

- “GETNUMPARAMETERS” on page 374
- “GETPARAMETERINFO” on page 374


```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE          PIC S9(8) BINARY.
   01 STMT-HANDLE         PIC S9(8) BINARY.
   01 PARAM-INDEX         PIC S9(8) BINARY.
   01 NATIVE-TYPE         PIC S9(8) BINARY.
   01 VAR                 PIC .....
   01 VAR-LEN             PIC S9(8) BINARY.
   01 VAR-IND             PIC S9(8) BINARY.
   01 OPTION              PIC ....
   01 OPT-LEN             PIC S9(8) BINARY.
   01 RETCODE             PIC S9(8) BINARY.
```

```
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-BINDPARAMETER ENV-HANDLE
        STMT-HANDLE PARM-INDEX NATIVE-TYPE VAR VAR-LEN VAR-IND
        [OPTION [OPT-LEN]] RETCODE.
```

FUNCTION (input)
A 16-byte character field containing BINDPARAMETER. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

STMT-HANDLE (input)
A fullword binary variable containing the statement handle.

PARAM-INDEX (input)
A fullword binary variable containing the number of the statement parameter to bind. Parameter numbers start at 1.

NATIVE-TYPE (input)
A fullword binary variable containing the native type of the host variable bound to the statement parameter. A native type value of 0 causes the parameter to be unbound. The following native types are available:

NATIVE-TYPE-STRING (1)
A textual string

NATIVE-TYPE-INTEGER-SIGNED (2)
A signed binary integer.

NATIVE-TYPE-INTEGER-UNSIGNED (3)
An unsigned binary integer.

NATIVE-TYPE-PACKED-SIGNED (4)
A signed packed decimal (COMP-3).

NATIVE-TYPE-PACKED-UNSIGNED (5)
An unsigned packed decimal.

NATIVE-TYPE-ZONED-SIGNED (6)
A signed zoned decimal.

NATIVE-TYPE-ZONED-UNSIGNED (7)
An unsigned zoned decimal.

NATIVE-TYPE-C-STRING (8)
A zero terminated string (C language string).

NATIVE-TYPE-FLOAT_BFP (9)
A floating point number in IEEE binary floating point representation (BFP).

NATIVE-TYPE-FLOAT_HFP (10)
A floating point number in hexadecimal floating point representation (HFP).

NATIVE-TYPE-TOD (11)
A 8 byte time of day clock (TOD) value.

NATIVE-TYPE-BOOLEAN (12)
A boolean value, i.e. a bit that is TRUE (1) or FALSE (0).

NATIVE-TYPE-BINARY (13)
A binary field.

NATIVE-TYPE-DATE_PACKED (32)
A date value in packed decimal format. Use the pattern option to specify the date pattern.

NATIVE-TYPE-TIME_PACKED (33)
A time value in packed decimal format. Use the pattern option to specify the time pattern.

NATIVE-TYPE-TIMESTAMP_PACKED (34)
A date and time value in packed decimal format. Use the pattern option to specify the date and time pattern.

NATIVE-TYPE-DATE_STRING (35)
A date value in text format. Use the pattern option to specify the date pattern.

NATIVE-TYPE-TIME_STRING (36)
A time value in text format. Use the pattern option to specify the time pattern.

NATIVE-TYPE-TIMESTAMP_STRING (37)
A date and time value in text format. Use the pattern option to specify the date and time pattern.

VAR (deferred input/output)
A host variable that is bound to the statement parameter. When the statement is executed the contents of the host variable is sent to the database. If the statement parameter denotes an output parameter, the execution of a stored procedure may also set the host variable. The parameter VAR-LEN specifies the length of the host variable in bytes. Use the COBOL term 'LENGTH OF' to determine the byte length of a variable.

When using REXX, this parameter must be a variable containing the bind variable name as a 32-character text string, padded with blanks. For this purpose, you can use the REXX function LEFT:

```
LEFT('variable-name',32)
```

VAR-LEN (input)
A fullword binary variable contains the length of the host variable in bytes. Use the COBOL term 'LENGTH OF' to determine the byte length of a variable.

VAR-IND (deferred input/output)
A fullword binary variable that acts as an indicator for the host variable bound to the statement parameter. The content of the indicator variable is determined when the statement is executed. It indicates if the variable value is NULL or not. If the parameter is an output parameter, the indicator variable may be updated when a stored procedure has been executed.

When using REXX, this parameter must be a variable containing the bind variable name as a 32-character text string, padded with blanks. For this purpose, you can use the REXX function LEFT:

```
LEFT('variable-name',32)
```

The indicator variable can contain the following values:

INDICATE-NOTNULL (0)
The variable value is not NULL. The content of the host variable bound to the parameter is used.

INDICATE-NULL (1)
The variable value is NULL.

OPTION (input, optional)
A variable that is used to specify an option value for a host variable. This parameter is optional. The following native types accept an option value:

NATIVE-TYPE-STRING and NATIVE-TYPE-C-STRING
The option value is a textual field specifying the EBCDIC codepage that is used to convert the host variable content from EBCDIC to ASCII and vice versa. The OPT-LEN parameter must be specified and must contain the length of the textual field.

All numeric types (PACKED, ZONED, INTEGER)
The option value is a fullword binary variable containing the implied decimal position used for the numeric data. The OPT-LEN parameter is not required for this option.

NATIVE-TYPE-BOOLEAN
The option value is a fullword binary variable containing the bit position counted from the right. Default is bit zero (right most bit). The OPT-LEN parameter is not required for this option.

All DATE, TIME and TIMESAMP types
The option value is a textual field specifying the pattern that is used to convert the host variable content into a database DATE, TIME or TIMESTAMP value and vice versa. Sample pattern format: "yyyy.MM.dd hh:mm:ss" will accept input like "1996.07.10 15:08:56". The OPT-LEN parameter must be specified and must contain the length of the textual field.

OPT-LEN (input, optional)
A fullword binary variable containing the length of the OPTION parameter if the OPTION is a textual field. This parameter is optional. If it is specified, the OPTION parameter must also be specified.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

CLOSECURSOR

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The CLOSECURSOR function closes an open cursor. After the cursor has been closed, you can no longer use the FETCH function to fetch data from the cursor. The close cursor also frees the memory allocated for the prefetch buffer.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE            PIC S9(8) BINARY.
   01 STMT-HANDLE           PIC S9(8) BINARY.
   01 RETCODE               PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-CLOSECURSOR ENV-HANDLE
        STMT-HANDLE RETCODE.
```

FUNCTION (input)
A 16-byte character field containing CLOSECURSOR. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

STMT-HANDLE (input)
A fullword binary variable containing the statement handle.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

CLOSESTATEMENT

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The CLOSESTATEMENT function closes the specified statement. After a successful call to the CLOSESTATEMENT function, the statement handle is no longer valid.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 STMT-HANDLE        PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-CLOSESTATEMENT ENV-HANDLE
        STMT-HANDLE RETCODE.
```

FUNCTION (input)
A 16-byte character field containing CLOSESTATEMENT. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

STMT-HANDLE (input)
A fullword binary variable containing the statement handle.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

COMMIT

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The COMMIT function ends a logical unit of work and commits all changes in the database performed so far on this connection or since the last commit. Dependent on its holdability, the commit call may close the cursor of a statement (see HOLD-CURSORS-OVER-COMMIT or CLOSE-CURSORS-AT-COMMIT values when preparing a statement).

Related function: "ROLLBACK" on page 385.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 CON-HANDLE         PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-COMMIT ENV-HANDLE CON-HANDLE
        RETCODE.
```

FUNCTION (input)
A 16-byte character field containing COMMIT. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

CONNECT

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The CONNECT function establishes a connection to the DBCliServer and to a database. It allocates a connection handle that represents the connection and sets the CON-HANDLE parameter. The application must pass the connection handle to all subsequent functions that require a connection. An application can establish multiple connections at a time. The application must ensure it uses the required connection handle with subsequent functions.

Related function:

- "CONNECTSSL" on page 311
- "DISCONNECT" on page 350

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE          PIC S9(8) BINARY.
   01 CON-HANDLE          PIC S9(8) BINARY.
   01 SERVER              PIC X(nnn) VALUE IS 'myhost.domain.com'.
   01 SERVER-LEN          PIC S9(8) BINARY VALUE IS 17.
   01 PORT                PIC S9(8) BINARY VALUE IS 16178.
   01 DBNAME              PIC X(nnn) VALUE IS 'SAMPLE'.
   01 DBNAME-LEN          PIC S9(8) BINARY VALUE IS 6.
   01 USERID              PIC X(nnn) VALUE IS 'DBUSER'.
   01 USERID-LEN          PIC S9(8) BINARY VALUE IS 6.
   01 PASSWD              PIC X(nnn) VALUE IS 'password'.
   01 PASSWD-LEN          PIC S9(8) BINARY VALUE IS 8.
   01 KEYNAME             PIC X(nnn) VALUE IS 'SAMPLE'.
   01 KEYNAME-LEN         PIC S9(8) BINARY VALUE IS 6.
   01 CIPHER              PIC X(nnn) VALUE IS '08090A62'.
   01 CIPHER-LEN          PIC S9(8) BINARY VALUE IS 8.
   01 RETCODE             PIC S9(8) BINARY.
PROCEDURE
    CALL 'IESDBCLI' USING FUNC-CONNECT ENV-HANDLE CON-HANDLE
         SERVER SERVER-LEN PORT
         DBNAME DBNAME-LEN [USERID USERID-LEN PASSWD PASSWD-LEN]
         RETCODE.
```

FUNCTION (input)
A 16-byte character field containing CONNECT. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (output)
A fullword binary variable containing the connection handle. The CON-HANDLE value set by the CONNECT call must be specified with subsequent call to other DBCLI functions that require a connection.

SERVER (input)
A character field containing the hostname, IPv4 or IPv6 address of DBCliServer to connect to. The field should be padded with blanks to the right, up to the length specified by the SERVER-LEN parameter.

SERVER-LEN (input)
A fullword binary variable containing the length of the hostname, IPv4 or IPv6 address specified in the SERVER parameter.

PORT (input)
A fullword binary variable containing the port number that will be used to connect to DBCliServer. Valid port numbers are from 1 to 65535. If you specify 0, the default port number is used. You can set the default port number using the environment attribute ENVATTR-DEFAULT-PORT.

DBNAME (input)
A character field containing the database name to connect to. The database name references to the databases configured in DBCliServer side. If the database name starts with jdbc: then it denotes the JDBC url to use. The field should be padded with blanks to the right, up to the length specified by the DBNAME-LEN parameter.

DBNAME-LEN (input)
A fullword binary variable containing the length of the database name specified in the DBNAME parameter.

USERID (input, optional)
An optional character field containing the user id used when connecting to the database. The field should be padded with blanks to the right, up to the length specified by the USERID-LEN parameter. If this parameter is specified, the parameters USERID-LEN, PASSWD and PASSWD-LEN must also be specified.

USERID-LEN (input, optional)
A fullword binary variable containing the length of the user id specified in the USERID parameter.

PASSWD (input, optional)
An optional character field containing the password used when connecting to the database. The field should be padded with blanks to the right, up to the length specified by the PASSWD-LEN parameter. If this parameter is specified, the parameters USERID, USERID-LEN and PASSWD-LEN must also be specified.

PASSWD-LEN (input, optional)
A fullword binary variable containing the length of the password specified in the PASSWD parameter.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

CONNECTSSL

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The CONNECTSSL function establishes a connection to the DBCLI server and to a database. It allocates a connection handle that represents the connection and sets the CON-HANDLE parameter. The application must pass the connection handle to all subsequent functions that require a connection. An application can establish multiple connections at a time. The application must ensure it uses the required connection handle with subsequent functions. CONNECTSSL establishes an *encrypted connection* to the DBCLI server (DBCliServer) using the SSL or TLS protocol. If SSL is used, you must call the INITSSL function before calling the CONNECTSSL function.

Related function:

- "CONNECT" on page 309
- "DISCONNECT" on page 350

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE           PIC S9(8) BINARY.
   01 CON-HANDLE           PIC S9(8) BINARY.
   01 SERVER               PIC X(nnn) VALUE IS 'myhost.domain.com'.
   01 SERVER-LEN           PIC S9(8) BINARY VALUE IS 17.
   01 PORT                 PIC S9(8) BINARY VALUE IS 16178.
   01 DBNAME               PIC X(nnn) VALUE IS 'SAMPLE'.
   01 DBNAME-LEN           PIC S9(8) BINARY VALUE IS 6.
   01 USERID               PIC X(nnn) VALUE IS 'DBUSER'.
   01 USERID-LEN           PIC S9(8) BINARY VALUE IS 6.
   01 PASSWD               PIC X(nnn) VALUE IS 'password'.
   01 PASSWD-LEN           PIC S9(8) BINARY VALUE IS 8.
   01 KEYNAME              PIC X(nnn) VALUE IS 'SAMPLE'.
   01 KEYNAME-LEN          PIC S9(8) BINARY VALUE IS 6.
   01 CIPHER               PIC X(nnn) VALUE IS '08090A62'.
   01 CIPHER-LEN           PIC S9(8) BINARY VALUE IS 8.
   01 RETCODE              PIC S9(8) BINARY.
PROCEDURE
    CALL 'IESDBCLI' USING FUNC-CONNECTSSL ENV-HANDLE CON-HANDLE
         SERVER SERVER-LEN PORT
         KEYNAME KEYNAME-LEN CIPHER CIPHER-LEN
         DBNAME DBNAME-LEN [USERID USERID-LEN PASSWD PASSWD-LEN]
         RETCODE.
```

FUNCTION (input)
A 16-byte character field containing CONNECTSSL. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (output)
A fullword binary variable containing the connection handle. The CON-HANDLE value set by the CONNECTSSL call must be specified with subsequent call to other DBCLI functions that require a connection.

SERVER (input)
A character field containing the hostname, IPv4 or IPv6 address of DBCliServer to connect to. The field should be padded with blanks to the right, up to the length specified by the SERVER-LEN parameter.

SERVER-LEN (input)
A fullword binary variable containing the length of the hostname, IPv4 or IPv6 address specified in the SERVER parameter.

PORT (input)
A fullword binary variable containing the port number that will be used to connect to DBCliServer. Valid port numbers are from 1 to 65535. If you specify 0, the default port number is used. You can set the default port number using the environment attribute ENVATTR-DEFAULT-PORT.

DBNAME (input)
A character field containing the database name to connect to. The database name references to the databases configured in DBCliServer side. If the database name starts with jdbc: then it denotes the JDBC url to use. The field should be padded with blanks to the right, up to the length specified by the DBNAME-LEN parameter.

DBNAME-LEN (input)
A fullword binary variable containing the length of the database name specified in the DBNAME parameter.

USERID (input, optional)
An optional character field containing the user id used when connecting to the database. The field should be padded with blanks to the right, up to the length specified by the USERID-LEN parameter. If this parameter is specified, the parameters USERID-LEN, PASSWD and PASSWD-LEN must also be specified.

USERID-LEN (input, optional)
A fullword binary variable containing the length of the user id specified in the USERID parameter.

PASSWD (input, optional)
An optional character field containing the password used when connecting to the database. The field should be padded with blanks to the right, up to the length specified by the PASSWD-LEN parameter. If this parameter is specified, the parameters USERID, USERID-LEN and PASSWD-LEN must also be specified.

PASSWD-LEN (input, optional)
A fullword binary variable containing the length of the password specified in the PASSWD parameter.

KEYNAME (input)
A character field containing the name of the SSL key to use with this connection. The field should be padded with blanks to the right, up to the length specified by the KEYNAME-LEN parameter.

KEYNAME-LEN (input)
A fullword binary variable containing the length of the key name specified in the KEYNAME parameter.

CIPHER (input)
A character field containing the SSL ciphers in order of usage preference to be used with this connection. The field should be padded with blanks to the right, up to the length specified by the CIPHER-LEN parameter. Values as supported by TCP/IP for VSE are:

```
01 for RSA512_NULL_MD5
02 for RSA512_NULL_SHA
08 for RSA512_DES40CBC_SHA
09 for RSA1024_DESCBC_SHA
0A for RSA1024_3DESCBC_SHA
62 for RSA1024_EXPORT_DESCBC_SHA
```

You can use any combination of these values in any order.

CIPHER-LEN (input)
A fullword binary variable containing the length of the ciphers specified in the CIPHER parameter.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

DBATTRIBUTES

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The DBATTRIBUTES function retrieves a description of the given attribute of the given type for a user-defined type (UDT) that is available in the given schema and catalog. It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like BINDCOLUMN and FETCH. The cursor must be closed using the CLOSECURSOR functions, and the statement must be closed using the CLOSESTATEMENT function.

Descriptions are returned only for attributes of UDTs matching the catalog, schema, type, and attribute name criteria. They are ordered by TYPE_SCHEM, TYPE_NAME and ORDINAL_POSITION. This description does not contain inherited attributes.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE           PIC S9(8) BINARY.
   01 CON-HANDLE           PIC S9(8) BINARY.
   01 STMT-HANDLE          PIC S9(8) BINARY.
   01 CATALOG              PIC X(nnn).
   01 CATALOG-LEN          PIC S9(8) BINARY.
   01 SCHEMAPATT           PIC X(nnn).
   01 SCHEMAPATT-LEN       PIC S9(8) BINARY.
   01 TYPENAMEPATT         PIC X(nnn).
   01 TYPENAMEPATT-LEN     PIC S9(8) BINARY.
   01 ATTRNAMEPATT         PIC X(nnn).
   01 ATTRNAMEPATT-LEN     PIC S9(8) BINARY.
   01 RETCODE              PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBATTRIBUTES ENV-HANDLE CON-HANDLE
        STMT-HANDLE CATALOG CATALOG-LEN SCHEMAPATT SCHEMAPATT-LEN
        TYPENAMEPATT TYPENAMEPATT-LEN ATTRNAMEPATT ATTRNAMEPATT-LEN
        RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DBATTRIBUTES. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

CATALOG (input)
A character field containing the catalog name. The field should be padded with

blanks to the right, up to the length specified by the CATALOG-LEN parameter. If CATALOG consists of blanks (but CATALOG-LEN is greater than zero), the function retrieves those without a catalog. If CATALOG-LEN is zero, the catalog name is not used to narrow the search.

CATALOG-LEN (input)
A fullword binary variable containing the length of the catalog name specified in the CATALOG parameter.

SCHEMAPATT (input)
A character field containing the schema name pattern. The field should be padded with blanks to the right, up to the length specified by the SCHEMAPATT-LEN parameter. If SCHEMAPATT consists of blanks (but SCHEMAPATT-LEN is greater than zero), the function retrieves those without a schema. If SCHEMAPATT-LEN is zero, the schema name pattern is not used to narrow the search. Within this pattern:

- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

SCHEMAPATT-LEN (input)
A fullword binary variable containing the length of the schema name pattern specified in the SCHEMAPATT parameter.

TYPENAMEPATT (input)
A character field containing the type name pattern. The field should be padded with blanks to the right, up to the length specified by the TYPENAMEPATT-LEN parameter. If TYPENAMEPATT-LEN is zero, the type name pattern is not used to narrow the search. Within this pattern:

- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

TYPENAMEPATT-LEN (input)
A fullword binary variable containing the length of the type name pattern specified in the TYPENAMEPATT parameter.

ATTRNAMEPATT (input)
A character field containing the attribute name pattern. The field should be padded with blanks to the right, up to the length specified by the ATTRNAMEPATT-LEN parameter. If ATTRNAMEPATT-LEN is zero, the attribute name pattern is not used to narrow the search. Within this pattern:

- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

ATTRNAMEPATT-LEN (input)
A fullword binary variable containing the length of the attribute name pattern specified in the ATTRNAMEPATT parameter.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:

1. `TYPE_CAT` (String): type catalog (may be NULL)
2. `TYPE_SCHEM` (String): type schema (may be NULL)
3. `TYPE_NAME` (String): type name
4. `ATTR_NAME` (String): attribute name
5. `DATA_TYPE` (int): attribute type SQL type

6. ATTR_TYPE_NAME (String): Data source dependent type name. For a UDT, the type name is fully qualified. For a REF, the type name is fully qualified and represents the target type of the reference type.
7. ATTR_SIZE (int): column size. For char or date types this is the maximum number of characters; for numeric or decimal types this is precision.
8. DECIMAL_DIGITS (int): the number of fractional digits
9. NUM_PREC_RADIX (int): Radix (typically either 10 or 2)
10. NULLABLE (int): whether NULL is allowed
    - NULLABLE-NONULLS (0): might not allow NULL values
    - NULLABLE-NULLABLE(1): definitely allows NULL values
    - NULLABLE-UNKNOWN(2): nullability unknown
11. REMARKS (String): comment describing column (may be NULL)
12. ATTR_DEF (String): default value (may be NULL)
13. SQL_DATA_TYPE (int): unused
14. SQL_DATETIME_SUB (int): unused
15. CHAR_OCTET_LENGTH (int): for char types the maximum number of bytes in the column
16. ORDINAL_POSITION (int): index of column in table (starting at 1)
17. IS_NULLABLE (String): "NO" means column definitely does not allow NULL values; "YES" means the column might allow NULL values. An empty string means unknown.
18. SCOPE_CATALOG (String): catalog of table that is the scope of a reference attribute (null if DATA_TYPE isn't REF)
19. SCOPE_SCHEMA (String): schema of table that is the scope of a reference attribute (null if DATA_TYPE isn't REF)
20. SCOPE_TABLE (String): table name that is the scope of a reference attribute (null if the DATA_TYPE isn't REF)
21. SOURCE_DATA_TYPE (short): source type of a distinct type or user-generated Ref type,SQL type from java.sql.Types (null if DATA_TYPE isn't DISTINCT or user-generated REF)

DBBESTROWIDENT

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The DBBESTROWIDENT function retrieves a description of a table's optimal set of columns that uniquely identifies a row. They are ordered by SCOPE. It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like BINDCOLUMN and FETCH. The cursor must be closed using the CLOSECURSOR functions, and the statement must be closed using the CLOSESTATEMENT function.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 CON-HANDLE         PIC S9(8) BINARY.
   01 STMT-HANDLE        PIC S9(8) BINARY.
   01 CATALOG            PIC X(64).
   01 CATALOG-LEN        PIC S9(8) BINARY.
   01 SCHEMA             PIC X(nnn).
   01 SCHEMA-LEN         PIC S9(8) BINARY.
   01 TABLENAME          PIC X(nnn).
```

```
   01 TABLENAME-LEN      PIC S9(8) BINARY.
   01 SCOPE              PIC S9(8) BINARY.
   01 NULLABLE           PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBBESTROWIDENT ENV-HANDLE CON-HANDLE
        STMT-HANDLE CATALOG CATALOG-LEN SCHEMA SCHEMA-LEN
        TABLENAME TABLENAME-LEN SCOPE NULLABLE RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DBBESTROWIDENT. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

CATALOG (input)
A character field containing the catalog name. The field should be padded with blanks to the right, up to the length specified by the CATALOG-LEN parameter. If CATALOG consists of blanks (but CATALOG-LEN is greater than zero), the function retrieves those without a catalog. If CATALOG-LEN is zero, the catalog name is not used to narrow the search.

CATALOG-LEN (input)
A fullword binary variable containing the length of the catalog name specified in the CATALOG parameter.

SCHEMA (input)
A character field containing the schema name. The field should be padded with blanks to the right, up to the length specified by the SCHEMA-LEN parameter. If SCHEMA consists of blanks (but SCHEMA-LEN is greater than zero), the function retrieves those without a schema. If SCHEMA-LEN is zero, the schema name is not used to narrow the search.

SCHEMA-LEN (input)
A fullword binary variable containing the length of the schema name specified in the SCHEMA parameter.

TABLENAME (input)
A character field containing the table name. The field should be padded with blanks to the right, up to the length specified by the TABLENAME-LEN parameter.

TABLENAME-LEN (input)
A fullword binary variable containing the length of the table name specified in the TABLENAME parameter.

SCOPE (input)
A fullword binary variable containing the scope of interest. Possible values are:

1. BESTROW-TEMPORARY (0) - very temporary, while using row
2. BESTROW-TRANSACTION (1) - valid for remainder of current transaction
3. BESTROW-SESSION (2) - valid for remainder of current session

NULLABLE (input)
A fullword binary variable containing either BOOLEAN-FALSE (0) or BOOLEAN-TRUE (1). This parameter determines if columns that are nullable are included.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:

1. `SCOPE (short)`: actual scope of result
   - BESTROW-TEMPORARY (0) - very temporary, while using row
   - BESTROW-TRANSACTION (1) - valid for remainder of current transaction
   - BESTROW-SESSION (2) - valid for remainder of current session
2. `COLUMN_NAME (String)`: column name
3. `DATA_TYPE (int)`: SQL data type from java.sql.Types
4. `TYPE_NAME (String)`: Data source dependent type name, for a UDT the type name is fully qualified
5. `COLUMN_SIZE (int)`: precision
6. `BUFFER_LENGTH (int)`: not used
7. `DECIMAL_DIGITS (short)`: scale
8. `PSEUDO_COLUMN (short)`: is this a pseudo column like an Oracle ROWID
   - BESTROW-UNKNOWN (0) - may or may not be pseudo column
   - BESTROW-NOTPSEUDO (1) - is NOT a pseudo column
   - BESTROW-PSEUDO (2) - is a pseudo column

DBCATALOGS

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The `DBCATALOGS` function retrieves a list of catalog names available in this database. It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like `BINDCOLUMN` and `FETCH`. The cursor must be closed using the `CLOSECURSOR` functions, and the statement must be closed using the `CLOSESTATEMENT` function.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE        PIC S9(8) BINARY.
   01 CON-HANDLE        PIC S9(8) BINARY.
   01 STMT-HANDLE       PIC S9(8) BINARY.
   01 RETCODE           PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBCATALOGS ENV-HANDLE CON-HANDLE
        STMT-HANDLE RETCODE.
```

FUNCTION (input)
A 16-byte character field containing `DBCATALOGS`. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:

- TABLE_CAT (String): catalog name

DBCOLUMNPRIV

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The DBCOLUMNPRIV function retrieves a description of the access rights for a table's columns. It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like BINDCOLUMN and FETCH. The cursor must be closed using the CLOSECURSOR functions, and the statement must be closed using the CLOSESTATEMENT function. Only privileges matching the column name criteria are returned. They are ordered by COLUMN_NAME and PRIVILEGE.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 CON-HANDLE         PIC S9(8) BINARY.
   01 STMT-HANDLE        PIC S9(8) BINARY.
   01 CATALOG            PIC X(nnn).
   01 CATALOG-LEN        PIC S9(8) BINARY.
   01 SCHEMA             PIC X(nnn).
   01 SCHEMA-LEN         PIC S9(8) BINARY.
   01 TABLENAME          PIC X(nnn).
   01 TABLENAME-LEN      PIC S9(8) BINARY.
   01 COLNAMEPATT        PIC X(nnn).
   01 COLNAMEPATT-LEN    PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBCOLUMNPRIV ENV-HANDLE CON-HANDLE
        STMT-HANDLE CATALOG CATALOG-LEN SCHEMA SCHEMA-LEN
        TABLENAME TABLENAME-LEN COLNAMEPATT COLNAMEPATT-LEN RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DBCOLUMNPRIV. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

CATALOG (input)
A character field containing the catalog name. The field should be padded with blanks to the right, up to the length specified by the CATALOG-LEN parameter. If CATALOG consists of blanks (but CATALOG-LEN is greater than

zero), the function retrieves those without a catalog. If CATALOG-LEN is zero, the catalog name is not used to narrow the search.

CATALOG-LEN (input)
A fullword binary variable containing the length of the catalog name specified in the CATALOG parameter.

SCHEMA (input)
A character field containing the schema name. The field should be padded with blanks to the right, up to the length specified by the SCHEMA-LEN parameter. If SCHEMA consists of blanks (but SCHEMA-LEN is greater than zero), the function retrieves those without a schema. If SCHEMA-LEN is zero, the schema name is not used to narrow the search.

SCHEMA-LEN (input)
A fullword binary variable containing the length of the schema name specified in the SCHEMA parameter.

TABLENAME (input)
A character field containing the table name. The field should be padded with blanks to the right, up to the length specified by the TABLENAME-LEN parameter.

TABLENAME-LEN (input)
A fullword binary variable containing the length of the table name specified in the TABLENAME parameter.

COLNAMEPATT (input)
A character field containing the column name pattern. The field should be padded with blanks to the right, up to the length specified by the COLNAMEPATT-LEN parameter. If COLNAMEPATT-LEN is zero, the column name pattern is not used to narrow the search. Within this pattern:

- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

COLNAMEPATT-LEN (input)
A fullword binary variable containing the length of the column name pattern specified in the COLNAMEPATT parameter.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:

1. `TABLE_CAT` (String): table catalog (may be NULL)
2. `TABLE_SCHEM` (String): table schema (may be NULL)
3. `TABLE_NAME` (String): table name
4. `COLUMN_NAME` (String): column name
5. `GRANTOR` (String): grantor of access (may be NULL)
6. `GRANTEE` (String): grantee of access
7. `PRIVILEGE` (String): name of access (SELECT, INSERT, UPDATE, REFRENCES, ...)
8. `IS_GRANTABLE` (String): "YES" if grantee is permitted to grant to others; "NO" if not; null if unknown

DBCOLUMNS

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The DBCOLUMNS function retrieves a description of table columns available in the specified catalog. It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like BINDCOLUMN and FETCH. The cursor must be closed using the CLOSECURSOR functions, and the statement must be closed using the CLOSESTATEMENT function. Only column descriptions matching the catalog, schema, table and column name criteria are returned. They are ordered by TABLE_SCHEM, TABLE_NAME, and ORDINAL_POSITION.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 CON-HANDLE         PIC S9(8) BINARY.
   01 STMT-HANDLE        PIC S9(8) BINARY.
   01 CATALOG            PIC X(nnn).
   01 CATALOG-LEN        PIC S9(8) BINARY.
   01 SCHEMAPATT         PIC X(nnn).
   01 SCHEMAPATT-LEN     PIC S9(8) BINARY.
   01 TABLEPATT          PIC X(nnn).
   01 TABLEPATT-LEN      PIC S9(8) BINARY.
   01 COLNAMEPATT        PIC X(nnn).
   01 COLNAMEPATT-LEN    PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBCOLUMNS ENV-HANDLE CON-HANDLE
        STMT-HANDLE CATALOG CATALOG-LEN SCHEMAPATT SCHEMAPATT-LEN
        TABLEPATT TABLEPATT-LEN COLNAMEPATT COLNAMEPATT-LEN RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DBCOLUMNS. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

CATALOG (input)
A character field containing the catalog name. The field should be padded with blanks to the right, up to the length specified by the CATALOG-LEN parameter. If CATALOG consists of blanks (but CATALOG-LEN is greater than zero), the function retrieves those without a catalog. If CATALOG-LEN is zero, the catalog name is not used to narrow the search.

CATALOG-LEN (input)
A fullword binary variable containing the length of the catalog name specified in the CATALOG parameter.

SCHEMAPATT (input)
A character field containing the schema name pattern. The field should be padded with blanks to the right, up to the length specified by the SCHEMAPATT-LEN parameter. If SCHEMAPATT consists of blanks (but SCHEMAPATT-LEN is greater than zero), the function retrieves those without

a schema. If SCHEMAPATT-LEN is zero, the schema name pattern is not used to narrow the search. Within this pattern:
- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

SCHEMAPATT-LEN (input)
A fullword binary variable containing the length of the schema name pattern specified in the SCHEMAPATT parameter.

TABLEPATT (input)
A character field containing the table name. The field should be padded with blanks to the right, up to the length specified by the TABLEPATT-LEN parameter. If TABLEPATT-LEN is zero, the table name is not used to narrow the search. Within this pattern:
- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

TABLEPATT-LEN (input)
A fullword binary variable containing the length of the table name specified in the TABLEPATT parameter.

COLNAMEPATT (input)
A character field containing the column name pattern. The field should be padded with blanks to the right, up to the length specified by the COLNAMEPATT-LEN parameter. When the COLNAMEPATT-LEN is zero, the column name pattern is not used to narrow the search. Within this pattern:
- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

COLNAMEPATT-LEN (input)
A fullword binary variable containing the length of the column name pattern specified in the COLNAMEPATT parameter.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:
1. `TABLE_CAT` (String): table catalog (may be NULL)
2. `TABLE_SCHEM` (String): table schema (may be NULL)
3. `TABLE_NAME` (String): table name
4. `COLUMN_NAME` (String): column name
5. `DATA_TYPE` (int): data type SQL type
6. `TYPE_NAME` (String): Data source dependent type name. For a UDT, the type name is fully qualified.
7. `COLUMN_SIZE` (int): column size. For char or date types this is the maximum number of characters; for numeric or decimal types this is precision.
8. `BUFFER_LENGTH` is not used.
9. `DECIMAL_DIGITS` (int): the number of fractional digits
10. `NUM_PREC_RADIX` (int): Radix (typically either 10 or 2)
11. `NULLABLE` (int): whether NULL is allowed
    - NULLABLE-NONULLS (0): might not allow NULL values
    - NULLABLE-NULLABLE(1): definitely allows NULL values
    - NULLABLE-UNKNOWN(2): nullability unknown

12. REMARKS (String): comment describing column (may be NULL)
13. COLUMN_DEF (String): default value (may be NULL)
14. SQL_DATA_TYPE (int): unused
15. SQL_DATETIME_SUB (int): unused
16. CHAR_OCTET_LENGTH (int): for char types the maximum number of bytes in the column
17. ORDINAL_POSITION (int): index of column in table (starting at 1)
18. IS_NULLABLE (String): "NO" means column definitely does not allow NULL values; "YES" means the column might allow NULL values. An empty string means unknown.
19. SCOPE_CATALOG (String): catalog of table that is the scope of a reference attribute (null if DATA_TYPE isn't REF)
20. SCOPE_SCHEMA (String): schema of table that is the scope of a reference attribute (null if DATA_TYPE isn't REF)
21. SCOPE_TABLE (String): table name that is the scope of a reference attribute (null if the DATA_TYPE isn't REF)
22. SOURCE_DATA_TYPE (short): source type of a distinct type or user-generated Ref type,SQL type from java.sql.Types (null if DATA_TYPE isn't DISTINCT or user-generated REF)

DBCROSSREFERENCE

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The DBCROSSREFERENCE function retrieves a description of the foreign key columns in the given foreign key table that reference the primary key columns of the given primary key table (describe how one table imports another's key). It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like BINDCOLUMN and FETCH. The cursor must be closed using the CLOSECURSOR functions, and the statement must be closed using the CLOSESTATEMENT function. This function should normally return a single foreign key/primary key pair because most tables import a foreign key from a table only once. They are ordered by FKTABLE_CAT, FKTABLE_SCHEM, FKTABLE_NAME, and KEY_SEQ.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 CON-HANDLE         PIC S9(8) BINARY.
   01 STMT-HANDLE        PIC S9(8) BINARY.
   01 PRIMCATALOG        PIC X(nnn).
   01 PRIMCATALOG-LEN    PIC S9(8) BINARY.
   01 PRIMSCHEMA         PIC X(nnn).
   01 PRIMSCHEMA-LEN     PIC S9(8) BINARY.
   01 PRIMTABLE          PIC X(nnn).
   01 PRIMTABLE-LEN      PIC S9(8) BINARY.
   01 FORCATALOG         PIC X(nnn).
   01 FORCATALOG-LEN     PIC S9(8) BINARY.
   01 FORSCHEMA          PIC X(nnn).
   01 FORSCHEMA-LEN      PIC S9(8) BINARY.
   01 FORTABLE           PIC X(nnn).
   01 FORTABLE-LEN       PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBCROSSREFERENCE ENV-HANDLE CON-HANDLE
```

```
STMT-HANDLE PRIMCATALOG PRIMCATALOG-LEN PRIMSCHEMA PRIMSCHEMA-LEN
PRIMTABLE PRIMTABLE-LEN FORCATALOG FORCATALOG-LEN
FORSCHEMA FORSCHEMA-LEN FORTABLE FORTABLE-LEN RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DBCROSSREFERENCE. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

PRIMCATALOG (input)
A character field containing the primary catalog name. The field should be padded with blanks to the right, up to the length specified by the PRIMCATALOG-LEN parameter. If PRIMCATALOG consists of blanks (but PRIMCATALOG-LEN is greater than zero), the function retrieves those without a catalog. If PRIMCATALOG-LEN is zero, the primary catalog name is not used to narrow the search.

PRIMCATALOG-LEN (input)
A fullword binary variable containing the length of the catalog name specified in the PRIMCATALOG parameter.

PRIMSCHEMA (input)
A character field containing the primary schema name. The field should be padded with blanks to the right, up to the length specified by the PRIMSCHEMA-LEN parameter. If PRIMSCHEMA consists of blanks (but PRIMSCHEMA-LEN is greater than zero), the function retrieves those without a schema. If PRIMSCHEMA-LEN is zero, the primary schema name is not used to narrow the search.

PRIMSCHEMA-LEN (input)
A fullword binary variable containing the length of the primary schema name specified in the PRIMSCHEMA parameter.

PRIMTABLE (input)
A character field containing the primary table name. The field should be padded with blanks to the right, up to the length specified by the PRIMTABLE-LEN parameter.

PRIMTABLE-LEN (input)
A fullword binary variable containing the length of the primary table name specified in the PRIMTABLE parameter.

FORCATALOG (input)
A character field containing the foreign catalog name. The field should be padded with blanks to the right, up to the length specified by the FORCATALOG-LEN parameter. If FORCATALOG consists of blanks (but FORCATALOG-LEN is greater than zero), the function retrieves those without a catalog. If FORCATALOG-LEN is zero, the catalog name is not used to narrow the search.

FORCATALOG-LEN (input)
: A fullword binary variable containing the length of the foreign catalog name specified in the FORCATALOG parameter.

FORSCHEMA (input)
: A character field containing the foreign schema name. The field should be padded with blanks to the right, up to the length specified by the FORSCHEMA-LEN parameter. If FORSCHEMA consists of blanks (but FORSCHEMA-LEN is greater than zero), the function retrieves those without a schema. If FORSCHEMA-LEN is zero, the foreign schema name is not used to narrow the search.

FORSCHEMA-LEN (input)
: A fullword binary variable containing the length of the foreign schema name specified in the FORSCHEMA parameter.

FORTABLE (input)
: A character field containing the foreign table name. The field should be padded with blanks to the right, up to the length specified by the FORTABLE-LEN parameter.

FORTABLE-LEN (input)
: A fullword binary variable containing the length of the foreign table name specified in the FORTABLE parameter.

RETCODE (output)
: A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:

1. `PKTABLE_CAT` (String): primary key table catalog (may be NULL)
2. `PKTABLE_SCHEM` (String): primary key table schema (may be NULL)
3. `PKTABLE_NAME` (String): primary key table name
4. `PKCOLUMN_NAME` (String): primary key column name
5. `FKTABLE_CAT` (String): foreign key table catalog (may be NULL) being exported (may be NULL)
6. `FKTABLE_SCHEM` (String): foreign key table schema (may be NULL) being exported (may be NULL)
7. `FKTABLE_NAME` (String): foreign key table name being exported
8. `FKCOLUMN_NAME` (String): foreign key column name being exported
9. `KEY_SEQ` (short): sequence number within foreign key
10. `UPDATE_RULE` (short): What happens to foreign key when primary is updated:
    - IMPORTEDKEY-CASCADE (0) - change imported key to agree with primary key update
    - IMPORTEDKEY-RESTRICT (1) - same as IMPORTEDKEY-NOACTION (for ODBC 2.x compatibility)
    - IMPORTEDKEY-SETNULL (2) - change imported key to NULL if its primary key has been updated
    - IMPORTEDKEY-NOACTION (3) - do not allow update of primary key if the primary key has been imported
    - IMPORTEDKEY-SETDEFAULT (4) - change imported key to default values if its primary key has been updated
11. `DELETE_RULE` (short): What happens to the foreign key when primary is deleted.

- IMPORTEDKEY-CASCADE (0) - delete rows that import a deleted key
- IMPORTEDKEY-RESTRICT (1) - same as IMPORTEDKEY-NOACTION (for ODBC 2.x compatibility)
- IMPORTEDKEY-SETNULL (2) - change imported key to NULL if its primary key has been deleted
- IMPORTEDKEY-NOACTION (3) - do not allow delete of primary key if the primary key has been imported
- IMPORTEDKEY-SETDEFAULT (4) - change imported key to default values if its primary key has been deleted

12. `FK_NAME` (String): foreign key name (may be NULL)
13. `PK_NAME` (String): primary key name (may be NULL)
14. `DEFERRABILITY` (short): can the evaluation of foreign key constraints be deferred until commit
    - IMPORTEDKEY-INITIALLYDEFERRED (5) - see SQL92 for definition
    - IMPORTEDKEY-INITIALLYIMMEDIATE (6) - see SQL92 for definition
    - IMPORTEDKEY-NOTDEFERRABLE (7) - see SQL92 for definition

DBEXPORTEDKEYS

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The `DBEXPORTEDKEYS` function retrieves a description of the foreign key columns that reference the given table's primary key columns (the foreign keys exported by a table). It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like `BINDCOLUMN` and `FETCH`. The cursor must be closed using the `CLOSECURSOR` functions, and the statement must be closed using the `CLOSESTATEMENT` function. They are ordered by FKTABLE_CAT, FKTABLE_SCHEM, FKTABLE_NAME, and KEY_SEQ.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE           PIC S9(8) BINARY.
   01 CON-HANDLE           PIC S9(8) BINARY.
   01 STMT-HANDLE          PIC S9(8) BINARY.
   01 CATALOG              PIC X(nnn).
   01 CATALOG-LEN          PIC S9(8) BINARY.
   01 SCHEMA               PIC X(nnn).
   01 SCHEMA-LEN           PIC S9(8) BINARY.
   01 TABLENAME            PIC X(nnn).
   01 TABLENAME-LEN        PIC S9(8) BINARY.
   01 RETCODE              PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBEXPORTEDKEYS ENV-HANDLE CON-HANDLE
        STMT-HANDLE CATALOG CATALOG-LEN SCHEMA SCHEMA-LEN
        TABLENAME TABLENAME-LEN RETCODE.
```

FUNCTION (input)
A 16-byte character field containing `DBEXPORTEDKEYS`. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

CATALOG (input)
A character field containing the catalog name. The field should be padded with blanks to the right, up to the length specified by the CATALOG-LEN parameter. If CATALOG consists of blanks (but CATALOG-LEN is greater than zero), the function retrieves those without a catalog. If CATALOG-LEN is zero, the catalog name is not used to narrow the search.

CATALOG-LEN (input)
A fullword binary variable containing the length of the catalog name specified in the CATALOG parameter.

SCHEMA (input)
A character field containing the schema name. The field should be padded with blanks to the right, up to the length specified by the SCHEMA-LEN parameter. If SCHEMA consists of blanks (but SCHEMA-LEN is greater than zero), the function retrieves those without a schema. If SCHEMA-LEN is zero, the schema name is not used to narrow the search.

SCHEMA-LEN (input)
A fullword binary variable containing the length of the schema name specified in the SCHEMA parameter.

TABLENAME (input)
A character field containing the table name. The field should be padded with blanks to the right, up to the length specified by the TABLENAME-LEN parameter.

TABLENAME-LEN (input)
A fullword binary variable containing the length of the table name specified in the TABLENAME parameter.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:

1. PKTABLE_CAT (String): primary key table catalog (may be NULL)
2. PKTABLE_SCHEM (String): primary key table schema (may be NULL)
3. PKTABLE_NAME (String): primary key table name
4. PKCOLUMN_NAME (String): primary key column name
5. FKTABLE_CAT (String): foreign key table catalog (may be NULL) being exported (may be NULL)
6. FKTABLE_SCHEM (String): foreign key table schema (may be NULL) being exported (may be NULL)
7. FKTABLE_NAME (String): foreign key table name being exported
8. FKCOLUMN_NAME (String): foreign key column name being exported
9. KEY_SEQ (short): sequence number within foreign key
10. UPDATE_RULE (short): What happens to foreign key when primary is updated:
    - IMPORTEDKEY-CASCADE (0) - change imported key to agree with primary key update

- IMPORTEDKEY-RESTRICT (1) - same as IMPORTEDKEY-NOACTION (for ODBC 2.x compatibility)
- IMPORTEDKEY-SETNULL (2) - change imported key to NULL if its primary key has been updated
- IMPORTEDKEY-NOACTION (3) - do not allow update of primary key if the primary key has been imported
- IMPORTEDKEY-SETDEFAULT (4) - change imported key to default values if its primary key has been updated

11. DELETE_RULE (short): What happens to the foreign key when primary is deleted.
    - IMPORTEDKEY-CASCADE (0) - delete rows that import a deleted key
    - IMPORTEDKEY-RESTRICT (1) - same as IMPORTEDKEY-NOACTION (for ODBC 2.x compatibility)
    - IMPORTEDKEY-SETNULL (2) - change imported key to NULL if its primary key has been deleted
    - IMPORTEDKEY-NOACTION (3) - do not allow delete of primary key if the primary key has been imported
    - IMPORTEDKEY-SETDEFAULT (4) - change imported key to default values if its primary key has been deleted
12. FK_NAME (String): foreign key name (may be NULL)
13. PK_NAME (String): primary key name (may be NULL)
14. DEFERRABILITY (short): can the evaluation of foreign key constraints be deferred until commit
    - IMPORTEDKEY-INITIALLYDEFERRED (5) - see SQL92 for definition
    - IMPORTEDKEY-INITIALLYIMMEDIATE (6) - see SQL92 for definition
    - IMPORTEDKEY-NOTDEFERRABLE (7) - see SQL92 for definition

DBIMPORTEDKEYS

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The DBIMPORTEDKEYS function retrieves a description of the primary key columns that are referenced by a table's foreign key columns (the primary keys imported by a table). It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like BINDCOLUMN and FETCH. The cursor must be closed using the CLOSECURSOR functions, and the statement must be closed using the CLOSESTATEMENT function. They are ordered by FKTABLE_CAT, FKTABLE_SCHEM, FKTABLE_NAME, and KEY_SEQ.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 CON-HANDLE         PIC S9(8) BINARY.
   01 STMT-HANDLE        PIC S9(8) BINARY.
   01 CATALOG            PIC X(nnn).
   01 CATALOG-LEN        PIC S9(8) BINARY.
   01 SCHEMA             PIC X(nnn).
   01 SCHEMA-LEN         PIC S9(8) BINARY.
   01 TABLENAME          PIC X(nnn).
   01 TABLENAME-LEN      PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
```

```
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBIMPORTEDKEYS ENV-HANDLE CON-HANDLE
        STMT-HANDLE CATALOG CATALOG-LEN SCHEMA SCHEMA-LEN
        TABLENAME TABLENAME-LEN RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DBIMPORTEDKEYS. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

CATALOG (input)
A character field containing the catalog name. The field should be padded with blanks to the right, up to the length specified by the CATALOG-LEN parameter. If CATALOG consists of blanks (but CATALOG-LEN is greater than zero), the function retrieves those without a catalog. If CATALOG-LEN is zero, the catalog name is not used to narrow the search.

CATALOG-LEN (input)
A fullword binary variable containing the length of the catalog name specified in the CATALOG parameter.

SCHEMA (input)
A character field containing the schema name. The field should be padded with blanks to the right, up to the length specified by the SCHEMA-LEN parameter. If SCHEMA consists of blanks (but SCHEMA-LEN is greater than zero), the function retrieves those without a schema. If SCHEMA-LEN is zero, the schema name is not used to narrow the search.

SCHEMA-LEN (input)
A fullword binary variable containing the length of the schema name specified in the SCHEMA parameter.

TABLENAME (input)
A character field containing the table name. The field should be padded with blanks to the right, up to the length specified by the TABLENAME-LEN parameter.

TABLENAME-LEN (input)
A fullword binary variable containing the length of the table name specified in the TABLENAME parameter.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:

1. PKTABLE_CAT (String): primary key table catalog (may be NULL)
2. PKTABLE_SCHEM (String): primary key table schema (may be NULL)
3. PKTABLE_NAME (String): primary key table name
4. PKCOLUMN_NAME (String): primary key column name

5. FKTABLE_CAT (String): foreign key table catalog (may be NULL) being exported (may be NULL)
6. FKTABLE_SCHEM (String): foreign key table schema (may be NULL) being exported (may be NULL)
7. FKTABLE_NAME (String): foreign key table name being exported
8. FKCOLUMN_NAME (String): foreign key column name being exported
9. KEY_SEQ (short): sequence number within foreign key
10. UPDATE_RULE (short): What happens to foreign key when primary is updated:
    - IMPORTEDKEY-CASCADE (0) - change imported key to agree with primary key update
    - IMPORTEDKEY-RESTRICT (1) - same as IMPORTEDKEY-NOACTION (for ODBC 2.x compatibility)
    - IMPORTEDKEY-SETNULL (2) - change imported key to NULL if its primary key has been updated
    - IMPORTEDKEY-NOACTION (3) - do not allow update of primary key if the primary key has been imported
    - IMPORTEDKEY-SETDEFAULT (4) - change imported key to default values if its primary key has been updated
11. DELETE_RULE (short): What happens to the foreign key when primary is deleted.
    - IMPORTEDKEY-CASCADE (0) - delete rows that import a deleted key
    - IMPORTEDKEY-RESTRICT (1) - same as IMPORTEDKEY-NOACTION (for ODBC 2.x compatibility)
    - IMPORTEDKEY-SETNULL (2) - change imported key to NULL if its primary key has been deleted
    - IMPORTEDKEY-NOACTION (3) - do not allow delete of primary key if the primary key has been imported
    - IMPORTEDKEY-SETDEFAULT (4) - change imported key to default values if its primary key has been deleted
12. FK_NAME (String): foreign key name (may be NULL)
13. PK_NAME (String): primary key name (may be NULL)
14. DEFERRABILITY (short): can the evaluation of foreign key constraints be deferred until commit
    - IMPORTEDKEY-INITIALLYDEFERRED (5) - see SQL92 for definition
    - IMPORTEDKEY-INITIALLYIMMEDIATE (6) - see SQL92 for definition
    - IMPORTEDKEY-NOTDEFERRABLE (7) - see SQL92 for definition

DBINDEXINFO

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The `DBINDEXINFO` function retrieves a description of the given table's indices and statistics. They are ordered by NON_UNIQUE, TYPE, INDEX_NAME, and ORDINAL_POSITION. It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like `BINDCOLUMN` and `FETCH`. The cursor must be closed using the `CLOSECURSOR` functions, and the statement must be closed using the `CLOSESTATEMENT` function.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 CON-HANDLE         PIC S9(8) BINARY.
   01 STMT-HANDLE        PIC S9(8) BINARY.
   01 CATALOG            PIC X(nnn).
   01 CATALOG-LEN        PIC S9(8) BINARY.
   01 SCHEMA             PIC X(nnn).
   01 SCHEMA-LEN         PIC S9(8) BINARY.
   01 TABLENAME          PIC X(nnn).
   01 TABLENAME-LEN      PIC S9(8) BINARY.
   01 UNIQUE             PIC S9(8) BINARY.
   01 APPROXIMATE        PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBINDEXINFO ENV-HANDLE CON-HANDLE
        STMT-HANDLE CATALOG CATALOG-LEN SCHEMA SCHEMA-LEN
        TABLENAME TABLENAME-LEN UNIQUE APPROXIMATE RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DBINDEXINFO. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

CATALOG (input)
A character field containing the catalog name. The field should be padded with blanks to the right, up to the length specified by the CATALOG-LEN parameter. If CATALOG consists of blanks (but CATALOG-LEN is greater than zero), the function retrieves those without a catalog. If CATALOG-LEN is zero, the catalog name is not used to narrow the search.

CATALOG-LEN (input)
A fullword binary variable containing the length of the catalog name specified in the CATALOG parameter.

SCHEMA (input)
A character field containing the schema name. The field should be padded with blanks to the right, up to the length specified by the SCHEMA-LEN parameter. If SCHEMA consists of blanks (but SCHEMA-LEN is greater than zero), the function retrieves those without a schema. If SCHEMA-LEN is zero, the schema name is not used to narrow the search.

SCHEMA-LEN (input)
A fullword binary variable containing the length of the schema name specified in the SCHEMA parameter.

TABLENAME (input)
A character field containing the table name. The field should be padded with blanks to the right, up to the length specified by the TABLENAME-LEN parameter.

TABLENAME-LEN (input)
A fullword binary variable containing the length of the table name specified in the TABLENAME parameter.

UNIQUE (input)
A fullword binary variable containing either BOOLEAN-FALSE (0) or BOOLEAN-TRUE (1). When true, return only indices for unique values; when false, return indices regardless of whether unique or not.

APPROXIMATE (input)
A fullword binary variable containing either BOOLEAN-FALSE (0) or BOOLEAN-TRUE (1). When true, result is allowed to reflect approximate or out of data values; when false, results are requested to be accurate.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:

1. `TABLE_CAT` (String): table catalog (may be NULL)
2. `TABLE_SCHEM` (String): table schema (may be NULL)
3. `TABLE_NAME` (String): table name
4. `NON_UNIQUE` (Boolean): Can index values be non-unique. false when TYPE is TABLEINDEX-STATISTIC (0)
5. `INDEX_QUALIFIER` (String): index catalog (may be NULL); null when TYPE is TABLEINDEX-STATISTIC (0)
6. `INDEX_NAME` (String): index name; null when TYPE is TABLEINDEX-STATISTIC (0)
7. `TYPE` (short): index type:
   - TABLEINDEX-STATISTIC (0) - this identifies table statistics that are returned in conjunction with a table's index descriptions
   - TABLEINDEX-CLUSTERED (1) - this is a clustered index
   - TABLEINDEX-HASHED (2) - this is a hashed index
   - TABLEINDEX-OTHER (3) - this is some other style of index
8. `ORDINAL_POSITION` (short): column sequence number within index; zero when TYPE is TABLEINDEX-STATISTIC (0)
9. `COLUMN_NAME` (String): column name; null when TYPE is TABLEINDEX-STATISTIC (0)
10. `ASC_OR_DESC` (String): column sort sequence, "A" = ascending, "D" = descending, may be NULL if sort sequence is not supported; null when TYPE is TABLEINDEX-STATISTIC (0)
11. `CARDINALITY` (int): If TYPE is TABLEINDEX-STATISTIC (0), this is the number of rows in the table; otherwise, it is the number of unique values in the index.
12. `PAGES` (int): If TYPE is TABLEINDEX-STATISTIC (0) then this is the number of pages used for the table, otherwise it is the number of pages used for thecurrent index.
13. `FILTER_CONDITION` (String): Filter condition, if any. (may be NULL)

DBPRIMARYKEYS

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The DBPRIMARYKEYS function retrieves description of the given table's primary key columns. They are ordered by COLUMN_NAME. It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like BINDCOLUMN and FETCH. The cursor must be closed using the CLOSECURSOR functions, and the statement must be closed using the CLOSESTATEMENT function.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 CON-HANDLE         PIC S9(8) BINARY.
   01 STMT-HANDLE        PIC S9(8) BINARY.
   01 CATALOG            PIC X(nnn).
   01 CATALOG-LEN        PIC S9(8) BINARY.
   01 SCHEMA             PIC X(nnn).
   01 SCHEMA-LEN         PIC S9(8) BINARY.
   01 TABLENAME          PIC X(nnn).
   01 TABLENAME-LEN      PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBPRIMARYKEYS ENV-HANDLE CON-HANDLE
        STMT-HANDLE CATALOG CATALOG-LEN SCHEMA SCHEMA-LEN
        TABLENAME TABLENAME-LEN RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DBPRIMARYKEYS. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

CATALOG (input)
A character field containing the catalog name. The field should be padded with blanks to the right, up to the length specified by the CATALOG-LEN parameter. If CATALOG consists of blanks (but CATALOG-LEN is greater than zero), the function retrieves those without a catalog. If CATALOG-LEN is zero, the catalog name is not used to narrow the search.

CATALOG-LEN (input)
A fullword binary variable containing the length of the catalog name specified in the CATALOG parameter.

SCHEMA (input)
A character field containing the schema name. The field should be padded with blanks to the right, up to the length specified by the SCHEMA-LEN parameter. If SCHEMA consists of blanks (but SCHEMA-LEN is greater than zero), the function retrieves those without a schema. If SCHEMA-LEN is zero, the schema name is not used to narrow the search.

SCHEMA-LEN (input)
A fullword binary variable containing the length of the schema name specified in the SCHEMA parameter.

TABLENAME (input)
A character field containing the table name. The field should be padded with blanks to the right, up to the length specified by the TABLENAME-LEN parameter.

TABLENAME-LEN (input)
A fullword binary variable containing the length of the table name specified in the TABLENAME parameter.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:

1. TABLE_CAT (String): table catalog (may be NULL)
2. TABLE_SCHEM (String): table schema (may be NULL)
3. TABLE_NAME (String): table name
4. COLUMN_NAME (String): column name
5. KEY_SEQ (short): sequence number within primary key
6. PK_NAME (String): primary key name (may be NULL)

DBPROCEDURECOLS

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The DBPROCEDURECOLS function retrieves a description of the given catalog's stored procedure parameter and result columns. It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like BINDCOLUMN and FETCH. The cursor must be closed using the CLOSECURSOR functions, and the statement must be closed using the CLOSESTATEMENT function.

Only descriptions that match the schema, procedure and parameter name criteria are returned. They are ordered by PROCEDURE_SCHEM and PROCEDURE_NAME. Within this, the return value, if any, is first. Next are the parameter descriptions in call order. The column descriptions follow in column number order.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 CON-HANDLE         PIC S9(8) BINARY.
   01 STMT-HANDLE        PIC S9(8) BINARY.
   01 CATALOG            PIC X(nnn).
   01 CATALOG-LEN        PIC S9(8) BINARY.
   01 SCHEMAPATT         PIC X(nnn).
   01 SCHEMAPATT-LEN     PIC S9(8) BINARY.
   01 PROCNAMEPATT       PIC X(nnn).
   01 PROCNAMEPATT-LEN   PIC S9(8) BINARY.
   01 COLNAMEPATT        PIC X(nnn).
   01 COLNAMEPATT-LEN    PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
PROCEDURE
```

```
CALL 'IESDBCLI' USING FUNC-DBPROCEDURECOLS ENV-HANDLE CON-HANDLE
     STMT-HANDLE CATALOG CATALOG-LEN SCHEMAPATT SCHEMAPATT-LEN
     PROCNAMEPATT PROCNAMEPATT-LEN COLNAMEPATT COLNAMEPATT-LEN RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DBPROCEDURECOLS. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

CATALOG (input)
A character field containing the catalog name. The field should be padded with blanks to the right, up to the length specified by the CATALOG-LEN parameter. If CATALOG consists of blanks (but CATALOG-LEN is greater than zero), the function retrieves those without a catalog. If CATALOG-LEN is zero, the catalog name is not used to narrow the search.

CATALOG-LEN (input)
A fullword binary variable containing the length of the catalog name specified in the CATALOG parameter.

SCHEMAPATT (input)
A character field containing the schema name pattern. The field should be padded with blanks to the right, up to the length specified by the SCHEMAPATT-LEN parameter. If SCHEMAPATT consists of blanks (but SCHEMAPATT-LEN is greater than zero), the function retrieves those without a schema. If SCHEMAPATT-LEN is zero, the schema name pattern is not used to narrow the search. Within this pattern:

- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

SCHEMAPATT-LEN (input)
A fullword binary variable containing the length of the schema name specified in the SCHEMAPATT parameter.

PROCNAMEPATT (input)
A character field containing the procedure name pattern. The field should be padded with blanks to the right, up to the length specified by the PROCNAMEPATT-LEN parameter. If PROCNAMEPATT-LEN is zero, the procedure name pattern is not used to narrow the search. Within this pattern:

- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

PROCNAMEPATT-LEN (input)
A fullword binary variable containing the length of the procedure name pattern specified in the PROCNAMEPATT parameter.

COLNAMEPATT (input)
A character field containing the column name pattern. The field should be padded with blanks to the right, up to the length specified by the COLNAMEPATT-LEN parameter. If COLNAMEPATT-LEN is zero, the column name pattern is not used to narrow the search. Within this pattern:

- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

COLNAMEPATT-LEN (input)
A fullword binary variable containing the length of the column name pattern specified in the COLNAMEPATT parameter.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:

1. PROCEDURE_CAT (String): procedure catalog (may be NULL)
2. PROCEDURE_SCHEM (String): procedure schema (may be NULL)
3. PROCEDURE_NAME (String): procedure name
4. COLUMN_NAME (String): column/parameter name
5. COLUMN_TYPE (short): kind of column/parameter:
   - PROCCOLUMN-UNKNOWN (0) - nobody knows
   - PROCCOLUMN-IN (1) - IN parameter
   - PROCCOLUMN-INOUT (2) - INOUT parameter
   - PROCCOLUMN-RESUILT (3) - result column in cursor
   - PROCCOLUMN-OUT (4) - OUT parameter
   - PROCCOLUMN-RETURN (5) - procedure return value
6. DATA_TYPE (int): data type SQL type
7. TYPE_NAME (String): Data source dependent type name. For a UDT, the type name is fully qualified.
8. PRECISION (int): precision
9. LENGTH (int): length in bytes of data
10. SCALE (short): scale
11. RADIX (short): radix
12. NULLABLE (short): can it contain NULL.
    - NULLABLE-NONULLS (0) - does not allow NULL values
    - NULLABLE-NULLABLE (1) - allows NULL values
    - NULLABLE-UNKNOWN (2) - nullability unknown
13. REMARKS (String): comment describing parameter/column

DBPROCEDURES

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The DBPROCEDURES function retrieves a description of the stored procedures available in the given catalog. It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like BINDCOLUMN and FETCH. The cursor must be closed using the CLOSECURSOR functions, and the statement must be closed using the CLOSESTATEMENT function. Only procedure descriptions matching the schema and procedure name criteria are returned. They are ordered by PROCEDURE_SCHEM and PROCEDURE_NAME.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE           PIC S9(8) BINARY.
```

```
   01 CON-HANDLE            PIC S9(8) BINARY.
   01 STMT-HANDLE           PIC S9(8) BINARY.
   01 CATALOG               PIC X(nnn).
   01 CATALOG-LEN           PIC S9(8) BINARY.
   01 SCHEMAPATT            PIC X(nnn).
   01 SCHEMAPATT-LEN        PIC S9(8) BINARY.
   01 PROCNAMEPATT          PIC X(nnn).
   01 PROCNAMEPATT-LEN      PIC S9(8) BINARY.
   01 RETCODE               PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBPROCEDURES ENV-HANDLE CON-HANDLE
        STMT-HANDLE CATALOG CATALOG-LEN SCHEMAPATT SCHEMAPATT-LEN
        PROCNAMEPATT PROCNAMEPATT-LEN RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DBPROCEDURES. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

CATALOG (input)
A character field containing the catalog name. The field should be padded with blanks to the right, up to the length specified by the CATALOG-LEN parameter. If CATALOG consists of blanks (but CATALOG-LEN is greater than zero), the function retrieves those without a catalog. If CATALOG-LEN is zero, the catalog name is not used to narrow the search.

CATALOG-LEN (input)
A fullword binary variable containing the length of the catalog name specified in the CATALOG parameter.

SCHEMAPATT (input)
A character field containing the schema name pattern. The field should be padded with blanks to the right, up to the length specified by the SCHEMAPATT-LEN parameter. If SCHEMAPATT consists of blanks (but SCHEMAPATT-LEN is greater than zero), the function retrieves those without a schema. If SCHEMAPATT-LEN is zero, the schema name pattern is not used to narrow the search. Within this pattern:

- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

SCHEMAPATT-LEN (input)
A fullword binary variable containing the length of the schema name pattern specified in the SCHEMAPATT parameter.

PROCNAMEPATT (input)
A character field containing the procedure name pattern. The field should be padded with blanks to the right, up to the length specified by the PROCNAMEPATT-LEN parameter. If PROCNAMEPATT-LEN is zero, the procedure name pattern is not used to narrow the search. Within this pattern:

- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

PROCNAMEPATT-LEN (input)
A fullword binary variable containing the length of the procedure name pattern specified in the PROCNAMEPATT parameter.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:

1. PROCEDURE_CAT (String): procedure catalog (may be NULL)
2. PROCEDURE_SCHEM (String): procedure schema (may be NULL)
3. PROCEDURE_NAME (String): procedure name
4. reserved for future use
5. reserved for future use
6. reserved for future use
7. REMARKS (String): explanatory comment on the procedure
8. PROCEDURE_TYPE (short): kind of procedure:
   - PROCTYPE-UNKNOWN (0) - May return a result
   - PROCTYPE-NORESULT (1) - Does not return a result
   - PROCTYPE-RETURNSRESULT (2) - Returns a result

DBSCHEMAS

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The DBSCHEMAS function retrieves a list of schema names available in this database. It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like BINDCOLUMN and FETCH. The cursor must be closed using the CLOSECURSOR functions, and the statement must be closed using the CLOSESTATEMENT function.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE           PIC S9(8) BINARY.
   01 CON-HANDLE           PIC S9(8) BINARY.
   01 STMT-HANDLE          PIC S9(8) BINARY.
   01 RETCODE              PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBSCHEMAS ENV-HANDLE CON-HANDLE
        STMT-HANDLE RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DBSCHEMAS. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:

1. TABLE_SCHEM (String): schema name
2. TABLE_CAT (String): catalog name (may be NULL)

DBSUPERTABLES

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The DBSUPERTABLES function retrieves a description of the table hierarchies defined in a particular schema in this database. It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like BINDCOLUMN and FETCH. The cursor must be closed using the CLOSECURSOR functions, and the statement must be closed using the CLOSESTATEMENT function. Only supertable information for tables matching the catalog, schema and table name are returned.

The table name parameter may be a fully- qualified name, in which case, the catalog and schema pattern parameters are ignored. If a table does not have a super table, it is not listed here. Supertables have to be defined in the same catalog and schema as the sub tables. Therefore, the type description does not need to include this information for the supertable. If the driver does not support type hierarchies, an empty result set is returned.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 CON-HANDLE         PIC S9(8) BINARY.
   01 STMT-HANDLE        PIC S9(8) BINARY.
   01 CATALOG            PIC X(nnn).
   01 CATALOG-LEN        PIC S9(8) BINARY.
   01 SCHEMAPATT         PIC X(nnn).
   01 SCHEMAPATT-LEN     PIC S9(8) BINARY.
   01 TABLENAMEPATT      PIC X(nnn).
   01 TABLENAMEPATT-LEN  PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBSUPERTABLES ENV-HANDLE CON-HANDLE
        STMT-HANDLE CATALOG CATALOG-LEN SCHEMAPATT SCHEMAPATT-LEN
        TABLENAMEPATT TABLENAMEPATT-LEN RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DBSUPERTABLES. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

CATALOG (input)
A character field containing the catalog name. The field should be padded with

blanks to the right, up to the length specified by the CATALOG-LEN parameter. If CATALOG consists of blanks (but CATALOG-LEN is greater than zero), the function retrieves those without a catalog. If CATALOG-LEN is zero, the catalog name is not used to narrow the search.

CATALOG-LEN (input)
A fullword binary variable containing the length of the catalog name specified in the CATALOG parameter.

SCHEMAPATT (input)
A character field containing the schema name pattern. The field should be padded with blanks to the right, up to the length specified by the SCHEMAPATT-LEN parameter. If SCHEMAPATT consists of blanks (but SCHEMAPATT-LEN is greater than zero), the function retrieves those without a schema. If SCHEMAPATT-LEN is zero, the schema name pattern is not used to narrow the search. Within this pattern:

- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

SCHEMAPATT-LEN (input)
A fullword binary variable containing the length of the schema name pattern specified in the SCHEMAPATT parameter.

TABLENAMEPATT (input)
A character field containing the table name pattern. The field should be padded with blanks to the right, up to the length specified by the TABLENAMEPATT-LEN parameter. If TABLENAMEPATT-LEN is zero, the table name pattern is not used to narrow the search. Within this pattern:

- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

TABLENAMEPATT-LEN (input)
A fullword binary variable containing the length of the table name pattern specified in the TABLENAMEPATT parameter.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:

- `TABLE_CAT` (String): table's catalog (may be NULL)
- `TABLE_SCHEM` (String): table's schema (may be NULL)
- `TABLE_NAME` (String): table name
- `SUPERTABLE_NAME` (String): the direct super table's name

DBSUPERTYPES

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The `DBSUPERTYPES` function retrieves a description of the user-defined type (UDT) hierarchies defined in a particular schema in this database. It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like `BINDCOLUMN` and `FETCH`. The cursor must be closed using the `CLOSECURSOR` functions, and the statement must be closed using the `CLOSESTATEMENT` function. Only the

immediate super type/ sub type relationship is modeled. Only supertype information for UDTs matching the catalog, schema, and type name is returned.

The type name parameter may be a fully-qualified name. If the UDT name supplied is a fully-qualified name, the catalog and schema pattern parameters are ignored. If a UDT does not have a direct super type, it is not listed here. A row of the cursor returned by this method describes the designated UDT and a direct supertype. If the driver does not support type hierarchies, an empty result set is returned.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE        PIC S9(8) BINARY.
   01 CON-HANDLE        PIC S9(8) BINARY.
   01 STMT-HANDLE       PIC S9(8) BINARY.
   01 CATALOG           PIC X(nnn).
   01 CATALOG-LEN       PIC S9(8) BINARY.
   01 SCHEMAPATT        PIC X(nnn).
   01 SCHEMAPATT-LEN    PIC S9(8) BINARY.
   01 TYPENAMEPATT      PIC X(nnn).
   01 TYPENAMEPATT-LEN  PIC S9(8) BINARY.
   01 RETCODE           PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBSUPERTYPES ENV-HANDLE CON-HANDLE
        STMT-HANDLE CATALOG CATALOG-LEN SCHEMAPATT SCHEMAPATT-LEN
        TYPENAMEPATT TYPENAMEPATT-LEN RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DBSUPERTYPES. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

CATALOG (input)
A character field containing the catalog name. The field should be padded with blanks to the right, up to the length specified by the CATALOG-LEN parameter. If CATALOG consists of blanks (but CATALOG-LEN is greater than zero), the function retrieves those without a catalog. If CATALOG-LEN is zero, the catalog name is not used to narrow the search.

CATALOG-LEN (input)
A fullword binary variable containing the length of the catalog name specified in the CATALOG parameter.

SCHEMAPATT (input)
A character field containing the schema name pattern. The field should be padded with blanks to the right, up to the length specified by the SCHEMAPATT-LEN parameter. If SCHEMAPATT consists of blanks (but SCHEMAPATT-LEN is greater than zero), the function retrieves those without a schema. If SCHEMAPATT-LEN is zero, the schema name pattern is not used to narrow the search. Within this pattern:

- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

SCHEMAPATT-LEN (input)
A fullword binary variable containing the length of the schema name pattern specified in the SCHEMAPATT parameter.

TYPENAMEPATT (input)
A character field containing the UDT name pattern. The field should be padded with blanks to the right, up to the length specified by the TYPENAMEPATT-LEN parameter. If TYPENAMEPATT-LEN is zero, the UDT name pattern is not used to narrow the search. Within this pattern:
- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

TYPENAMEPATT-LEN (input)
A fullword binary variable containing the length of the UDT name pattern specified in the TYPENAMEPATT parameter.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:
1. TYPE_CAT (String): the UDT's catalog (may be NULL)
2. TYPE_SCHEM (String): UDT's schema (may be NULL)
3. TYPE_NAME (String): type name of the UDT
4. SUPERTYPE_CAT (String): the direct super type's catalog (may be NULL)
5. SUPERTYPE_SCHEM (String): the direct super type's schema (may be NULL)
6. SUPERTYPE_NAME (String): the direct super type's name

DBTABLEPRIV

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The DBTABLEPRIV function retrieves a description of the access rights for each table available in a catalog. It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like BINDCOLUMN and FETCH. The cursor must be closed using the CLOSECURSOR functions, and the statement must be closed using the CLOSESTATEMENT function.

Note that a table privilege applies to one or more columns in the table:
- It would be wrong to assume that this privilege applies to all columns (this may be true for some systems but is not true for all).
- Only privileges matching the schema and table name criteria are returned. They are ordered by TABLE_SCHEM, TABLE_NAME, and PRIVILEGE.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE           PIC S9(8) BINARY.
   01 CON-HANDLE           PIC S9(8) BINARY.
   01 STMT-HANDLE          PIC S9(8) BINARY.
   01 CATALOG              PIC X(nnn).
   01 CATALOG-LEN          PIC S9(8) BINARY.
   01 SCHEMAPATT           PIC X(nnn).
   01 SCHEMAPATT-LEN       PIC S9(8) BINARY.
   01 TABLEPATT            PIC X(nnn).
   01 TABLEPATT-LEN        PIC S9(8) BINARY.
   01 RETCODE              PIC S9(8) BINARY.
```

```
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBTABLEPRIV ENV-HANDLE CON-HANDLE
        STMT-HANDLE CATALOG CATALOG-LEN SCHEMAPATT SCHEMAPATT-LEN
        TABLEPATT TABLEPATT-LEN RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DBTABLEPRIV. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

CATALOG (input)
A character field containing the catalog name. The field should be padded with blanks to the right, up to the length specified by the CATALOG-LEN parameter. If CATALOG consists of blanks (but CATALOG-LEN is greater than zero), the function retrieves those without a catalog. If CATALOG-LEN is zero, the catalog name is not used to narrow the search.

CATALOG-LEN (input)
A fullword binary variable containing the length of the catalog name specified in the CATALOG parameter.

SCHEMAPATT (input)
A character field containing the schema name pattern. The field should be padded with blanks to the right, up to the length specified by the SCHEMAPATT-LEN parameter. If SCHEMAPATT consists of blanks (but SCHEMAPATT-LEN is greater than zero), the function retrieves those without a schema. If SCHEMAPATT-LEN is zero, the schema name pattern is not used to narrow the search. Within this pattern:

- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

SCHEMAPATT-LEN (input)
A fullword binary variable containing the length of the schema name pattern specified in the SCHEMA parameter.

TABLEPATT (input)
A character field containing the table name pattern. The field should be padded with blanks to the right, up to the length specified by the TABLEPATT-LEN parameter. If TABLEPATT-LEN is zero, the table name pattern is not used to narrow the search. Within this pattern:

- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

TABLEPATT-LEN (input)
A fullword binary variable containing the length of the table name pattern specified in the TABLEPATT parameter.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:

1. TABLE_CAT (String): table catalog (may be NULL)
2. TABLE_SCHEM (String): table schema (may be NULL)
3. TABLE_NAME (String): table name
4. GRANTOR (String): grantor of access (may be NULL)
5. GRANTEE (String): grantee of access
6. PRIVILEGE (String): name of access (SELECT, INSERT, UPDATE, REFRENCES, ...)
7. IS_GRANTABLE (String): "YES" if grantee is permitted to grant to others; "NO" if not; null if unknown

DBTABLES

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The DBTABLES function retrieves a description of the tables available in the given catalog. It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like BINDCOLUMN and FETCH. The cursor must be closed using the CLOSECURSOR functions, and the statement must be closed using the CLOSESTATEMENT function.

Only table descriptions matching the catalog, schema, table name and type criteria are returned. They are ordered by TABLE_TYPE, TABLE_SCHEM and TABLE_NAME.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 CON-HANDLE         PIC S9(8) BINARY.
   01 STMT-HANDLE        PIC S9(8) BINARY.
   01 CATALOG            PIC X(nnn).
   01 CATALOG-LEN        PIC S9(8) BINARY.
   01 SCHEMAPATT         PIC X(nnn).
   01 SCHEMAPATT-LEN     PIC S9(8) BINARY.
   01 TABLEPATT          PIC X(nnn).
   01 TABLEPATT-LEN      PIC S9(8) BINARY.
   01 TYPES              PIC X(nnn).
   01 TYPES-LEN          PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBTABLES ENV-HANDLE CON-HANDLE
        STMT-HANDLE CATALOG CATALOG-LEN SCHEMAPATT SCHEMAPATT-LEN
        TABLEPATT TABLEPATT-LEN TYPES TYPES-LEN RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DBTABLES. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

CATALOG (input)
A character field containing the catalog name. The field should be padded with blanks to the right, up to the length specified by the CATALOG-LEN parameter. If CATALOG consists of blanks (but CATALOG-LEN is greater than zero), the function retrieves those without a catalog. If CATALOG-LEN is zero, the catalog name is not used to narrow the search.

CATALOG-LEN (input)
A fullword binary variable containing the length of the catalog name specified in the CATALOG parameter.

SCHEMAPATT (input)
A character field containing the schema name pattern. The field should be padded with blanks to the right, up to the length specified by the SCHEMAPATT-LEN parameter. If SCHEMAPATT consists of blanks (but SCHEMAPATT-LEN is greater than zero), the function retrieves those without a schema. If SCHEMAPATT-LEN is zero, the schema name pattern is not used to narrow the search. Within this pattern:

- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

SCHEMAPATT-LEN (input)
A fullword binary variable containing the length of the schema name pattern specified in the SCHEMAPATT parameter.

TABLEPATT (input)
A character field containing the table name pattern. The field should be padded with blanks to the right, up to the length specified by the TABLEPATT-LEN parameter. If TABLEPATT-LEN is zero, the table name pattern is not used to narrow the search. Within this pattern:

- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

TABLEPATT-LEN (input)
A fullword binary variable containing the length of the table name pattern specified in the TABLEPATT parameter.

TYPES (input)
A character field containing a semicolon separated list of type names. The field should be padded with blanks to the right, up to the length specified by TYPES-LEN parameter. If TYPES-LEN is zero, all types are included.

TYPES-LEN (input)
A fullword binary variable containing the length of the type list specified in the TYPES parameter.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:

1. `TABLE_CAT` (String): table catalog (may be NULL)
2. `TABLE_SCHEM` (String): table schema (may be NULL)
3. `TABLE_NAME` (String): table name
4. `TABLE_TYPE` (String): table type. Typical types are "TABLE", "VIEW", "SYSTEM TABLE", "GLOBAL TEMPORARY", "LOCAL TEMPORARY", "ALIAS", "SYNONYM".
5. `REMARKS` (String): explanatory comment on the table

6. TYPE_CAT (String): the types catalog (may be NULL)
7. TYPE_SCHEM (String): the types schema (may be NULL)
8. TYPE_NAME (String): type name (may be NULL)
9. SELF_REFERENCING_COL_NAME (String): name of the designated "identifier" column of a typed table (may be NULL)
10. REF_GENERATION (String): specifies how values in SELF_REFERENCING_COL_NAME are created. Values are "SYSTEM", "USER", "DERIVED". (May be NULL.)

DBTABLETYPES

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The DBTABLETYPES function retrieves a list of available table types. It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like BINDCOLUMN and FETCH. The cursor must be closed using the CLOSECURSOR functions, and the statement must be closed using the CLOSESTATEMENT function.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE            PIC S9(8) BINARY.
   01 CON-HANDLE            PIC S9(8) BINARY.
   01 STMT-HANDLE           PIC S9(8) BINARY.
   01 RETCODE               PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBTABLETYPES ENV-HANDLE CON-HANDLE
        STMT-HANDLE RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DBTABLETYPES. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:

- TABLE_TYPE (String): table type. Typical types are "TABLE", "VIEW", "SYSTEM TABLE", "GLOBAL TEMPORARY", "LOCAL TEMPORARY", "ALIAS", "SYNONYM".

DBTYPEINFO

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The DBTYPEINFO function retrieves a description of all the standard SQL types supported by this database. It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like BINDCOLUMN and FETCH. The cursor must be closed using the CLOSECURSOR functions, and the statement must be closed using the CLOSESTATEMENT function. The returned types are ordered by DATA_TYPE and then by how closely the data type maps to the corresponding JDBC SQL type.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 CON-HANDLE         PIC S9(8) BINARY.
   01 STMT-HANDLE        PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBTYPEINFO ENV-HANDLE CON-HANDLE
        STMT-HANDLE RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DBTYPEINFO. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:

1. TYPE_NAME (String): Type name
2. DATA_TYPE (int): SQL data type
3. PRECISION (int): maximum precision
4. LITERAL_PREFIX (String): prefix used to quote a literal (may be null)
5. LITERAL_SUFFIX (String): suffix used to quote a literal (may be null)
6. CREATE_PARAMS (String): parameters used in creating the type (may be null)
7. NULLABLE (short): can you use NULL for this type.
   - NULLABLE-NONULLS (0) - does not allow NULL values
   - NULLABLE-NULLABLE (1) - allows NULL values
   - NULLABLE-UNKNOWN (2) - nullability unknown
8. CASE_SENSITIVE (Boolean): is it case sensitive.
9. SEARCHABLE (short): can you use "WHERE" based on this type:
   - TYPEPRED-NONE (0) - No support
   - TYPEPRED-CHAR (1) - Only supported with WHERE .. LIKE

- TYPEPRED-BASIC (2) - Supported except for WHERE .. LIKE
- TYPEPRED-SEARCHABLE (3) - Supported for all WHERE ..

10. UNSIGNED_ATTRIBUTE (Boolean): is it unsigned.
11. FIXED_PREC_SCALE (Boolean): can it be a money value.
12. AUTO_INCREMENT (Boolean): can it be used for an auto-increment value.
13. LOCAL_TYPE_NAME (String): localized version of type name (may be null)
14. MINIMUM_SCALE (short): minimum scale supported
15. MAXIMUM_SCALE (short): maximum scale supported
16. SQL_DATA_TYPE (int): unused
17. SQL_DATETIME_SUB (int): unused
18. NUM_PREC_RADIX (int): usually 2 or 10

DBUDTS

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The DBUDTS function retrieves a description of the user-defined types (UDTs) defined in a particular schema. It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like BINDCOLUMN and FETCH. The cursor must be closed using the CLOSECURSOR functions, and the statement must be closed using the CLOSESTATEMENT function.

Schema-specific UDTs may have type JAVA_OBJECT, STRUCT, or DISTINCT. Only types matching the catalog, schema, type name and type criteria are returned. They are ordered by DATA_TYPE, TYPE_SCHEM and TYPE_NAME. The type name parameter may be a fully-qualified name. In this case, the catalog and schema pattern parameters are ignored. If the driver does not support UDTs, an empty result set is returned

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE        PIC S9(8) BINARY.
   01 CON-HANDLE        PIC S9(8) BINARY.
   01 STMT-HANDLE       PIC S9(8) BINARY.
   01 CATALOG           PIC X(nnn).
   01 CATALOG-LEN       PIC S9(8) BINARY.
   01 SCHEMAPATT        PIC X(nnn).
   01 SCHEMAPATT-LEN    PIC S9(8) BINARY.
   01 TYPEPATT          PIC X(nnn).
   01 TYPEPATT-LEN      PIC S9(8) BINARY.
   01 TYPES             PIC X(nnn).
   01 TYPES-LEN         PIC S9(8) BINARY.
   01 RETCODE           PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBUDTS ENV-HANDLE CON-HANDLE
        STMT-HANDLE CATALOG CATALOG-LEN SCHEMAPATT SCHEMAPATT-LEN
        TYPEPATT TYPEPATT-LEN TYPES TYPES-LEN RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DBUDTS. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

CATALOG (input)
A character field containing the catalog name. The field should be padded with blanks to the right, up to the length specified by the CATALOG-LEN parameter. If CATALOG consists of blanks (but CATALOG-LEN is greater than zero), the function retrieves those without a catalog. If CATALOG-LEN is zero, the catalog name is not used to narrow the search.

CATALOG-LEN (input)
A fullword binary variable containing the length of the catalog name specified in the CATALOG parameter.

SCHEMAPATT (input)
A character field containing the schema name pattern. The field should be padded with blanks to the right, up to the length specified by the SCHEMAPATT-LEN parameter. If SCHEMAPATT consists of blanks (but SCHEMAPATT-LEN is greater than zero), the function retrieves those without a schema. If SCHEMAPATT-LEN is zero, the schema name pattern is not used to narrow the search. Within this pattern:

- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

SCHEMAPATT-LEN (input)
A fullword binary variable containing the length of the schema name pattern specified in the SCHEMAPATT parameter.

TYPEPATT (input)
A character field containing the type name pattern. The field should be padded with blanks to the right, up to the length specified by the TYPEPATT-LEN parameter. If TYPEPATT-LEN is zero, the type name pattern is not used to narrow the search. Within this pattern:

- '%' means match any substring of 0 or more characters.
- '_' means match any one character.

TYPEPATT-LEN (input)
A fullword binary variable containing the length of the type name pattern specified in the TYPEPATT parameter.

TYPES (input)
A character field containing a semicolon separated list of type names. The field should be padded with blanks to the right, up to the length specified by TYPES-LEN parameter. If TYPES-LEN is zero, all types are included.

TYPES-LEN (input)
A fullword binary variable containing the length of the type list specified in the TYPES parameter.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:

1. `TYPE_CAT` (String): the type's catalog (may be NULL)
2. `TYPE_SCHEM` (String): type's schema (may be NULL)
3. `TYPE_NAME` (String): type name

4. CLASS_NAME (String): Java class name
5. DATA_TYPE (int): SQL type value. One of JAVA_OBJECT, STRUCT, or DISTINCT
6. REMARKS (String): explanatory comment on the type
7. BASE_TYPE (short): type code of the source type of a DISTINCT type or the type that implements the user-generated reference type of the SELF_REFERENCING_COLUMN of a structured type as defined in SQL Types (NULL if DATA_TYPE is not DISTINCT or not STRUCT with REFERENCE_GENERATION = USER_DEFINED)

DBVERSIONCOLS

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The DBVERSIONCOLS function retrieves a description of a table's columns that are automatically updated when any value in a row is updated. It allocates a statement handle and sets the STMT-HANDLE parameter. The statement contains an open cursor that has to be used to retrieve the returned information using functions like BINDCOLUMN and FETCH. The cursor must be closed using the CLOSECURSOR functions, and the statement must be closed using the CLOSESTATEMENT function. Only descriptions matching the schema, procedure and parameter name criteria are returned.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 CON-HANDLE         PIC S9(8) BINARY.
   01 STMT-HANDLE        PIC S9(8) BINARY.
   01 CATALOG            PIC X(nnn).
   01 CATALOG-LEN        PIC S9(8) BINARY.
   01 SCHEMA             PIC X(nnn).
   01 SCHEMA-LEN         PIC S9(8) BINARY.
   01 TABLENAME          PIC X(nnn).
   01 TABLENAME-LEN      PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DBVERSIONCOLS ENV-HANDLE CON-HANDLE
        STMT-HANDLE CATALOG CATALOG-LEN SCHEMA SCHEMA-LEN
        TABLENAME TABLENAME-LEN RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DBVERSIONCOLS. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by this function call must be specified with subsequent call to other DBCLI functions that require a statement.

CATALOG (input)
A character field containing the catalog name. The field should be padded with blanks to the right, up to the length specified by the CATALOG-LEN parameter. If CATALOG consists of blanks (but CATALOG-LEN is greater than zero), the function retrieves those without a catalog. If CATALOG-LEN is zero, the catalog name is not used to narrow the search.

CATALOG-LEN (input)
A fullword binary variable containing the length of the catalog name specified in the CATALOG parameter.

SCHEMA (input)
A character field containing the schema name. The field should be padded with blanks to the right, up to the length specified by the SCHEMA-LEN parameter. If SCHEMA consists of blanks (but SCHEMA-LEN is greater than zero), the function retrieves those without a schema. If SCHEMA-LEN is zero, the schema name is not used to narrow the search.

SCHEMA-LEN (input)
A fullword binary variable containing the length of the schema name specified in the SCHEMA parameter.

TABLENAME (input)
A character field containing the table name. The field should be padded with blanks to the right, up to the length specified by the TABLENAME-LEN parameter.

TABLENAME-LEN (input)
A fullword binary variable containing the length of the table name specified in the TABLENAME parameter.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The returned cursor contains the following columns:
1. SCOPE (short): is not used
2. COLUMN_NAME (String): column name
3. DATA_TYPE (int): SQL data type from java.sql.Types
4. TYPE_NAME (String): Data source-dependent type name
5. COLUMN_SIZE (int): precision
6. BUFFER_LENGTH (int): length of column value in bytes
7. DECIMAL_DIGITS (short): scale
8. PSEUDO_COLUMN (short): whether this is pseudo column like an Oracle ROWID
   - VERSIONCOL-UNKNOWN (0) - may or may not be pseudo column
   - VERSIONCOL-NOTPSEUDO (1) - is NOT a pseudo column
   - VERSIONCOL-PSEUDO (2) - is a pseudo column

DISCONNECT

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The DISCONNECT function closes a connection to DBCliServer and also disconnects from the database. It frees a connection handle that represents the connection. After a successful call to the DISCONNECT function, the connection handle is no longer valid.

Related functions:

- “CONNECT” on page 309
- “CONNECTSSL” on page 311

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE          PIC S9(8) BINARY.
   01 CON-HANDLE          PIC S9(8) BINARY.
   01 RETCODE             PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-DISCONNECT ENV-HANDLE CON-HANDLE
        RETCODE.
```

FUNCTION (input)
A 16-byte character field containing DISCONNECT. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

EXECUTE

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The EXECUTE function executes a statement. The content of any host variables bound to input parameters will be sent to the database and will replace the parameters in the statement.

- If the statement was an update, the update count is provided with this function, or can be retrieved at a later time using the GETUPDATECOUNT function.
- If the statement was a query that returns a cursor, the result rows can be fetched using the FETCH function.

A statement can provide multiple results. You must use the GETMORERESULTS function to retrieve additional results. To retrieve the output variables from a stored procedure, you must call the GETMORERESULTS function until no more results are available. Only then the host variables bound to output parameters will be set to the value set by the stored procedure. You can re-execute a statement (with different input parameter values) at any time. Re-execution automatically closes a cursor that may have been left open from the last execution.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE          PIC S9(8) BINARY.
   01 STMT-HANDLE         PIC S9(8) BINARY.
   01 UPDATE-COUNT        PIC S9(8) BINARY.
   01 RETCODE             PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-EXECUTE ENV-HANDLE
        STMT-HANDLE [UPDATE-COUNT] RETCODE.
```

FUNCTION (input)
A 16-byte character field containing EXECUTE. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

STMT-HANDLE (input)
A fullword binary variable containing the statement handle.

UPDATE-COUNT (ouput, optional)
A fullword binary variable that is set to the number of rows updated by the statement execution or -1 if no update count is available.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

FETCH

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The FETCH function allows an application to retrieve one row of data from an open cursor. Dependent on the fetch operation, the FETCH function positions the cursor to the required row. The FETCH function will set all host variables which have been bound to the cursor using the BINDCOLUMN function. It also sets the corresponding indicator variables to indicate if a column is NULL, or was updated, deleted or inserted. After a statement has been executed, the cursor is positioned before the first row.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE            PIC S9(8) BINARY.
   01 STMT-HANDLE           PIC S9(8) BINARY.
   01 OPERATION             PIC S9(8) BINARY.
   01 OFFSET                PIC S9(8) BINARY.
   01 RETCODE               PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-FETCH ENV-HANDLE
        STMT-HANDLE [OPERATION [OFFSET]] RETCODE.
```

FUNCTION (input)
A 16-byte character field containing FETCH. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

STMT-HANDLE (input)
A fullword binary variable containing the statement handle.

OPERATION (input, optional)
A fullword binary variable containing an operation code. If the OPERATION parameter is omitted, the operation defaults to FETCH-NEXT. The following operations are supported:

FETCH-NEXT (0)
Positions the cursor to the next row and fetches the data. This is the default operation.

FETCH-PREVIOUS (1)
Positions the cursor to the previous row and fetches the data. This operation is only available if the cursor is scrollable, that's if the statement has been created with CURSOR-TYPE-SCROLL-INSENSITIVE or CURSOR-TYPE-SCROLL-SENSITIVE.

FETCH-FIRST (2)
Positions the cursor to the first row of the result and fetches the data. This operation is only available if the cursor is scrollable, that's if the statement has been created with CURSOR-TYPE-SCROLL-INSENSITIVE or CURSOR-TYPE-SCROLL-SENSITIVE.

FETCH-LAST (3)
Positions the cursor to the last row of the result and fetches the data. This operation is only available if the cursor is scrollable, that's if the statement has been created with CURSOR-TYPE-SCROLL-INSENSITIVE or CURSOR-TYPE-SCROLL-SENSITIVE.

FETCH-ABSOLUTE (4)
Positions the cursor to the row specified in the OFFSET parameter. A row number of zero positions the cursor before the first row. A negative row number positions the cursor to the last row of the result. This operation is only available if the cursor is scrollable, that's if the statement has been created with CURSOR-TYPE-SCROLL-INSENSITIVE or CURSOR-TYPE-SCROLL-SENSITIVE.

FETCH-RELATIVE (5)
Positions the moves the cursor by the number of rows specified in the OFFSET parameter. A negative OFFSET moves the cursor towards the start of the result, a positive OFFSET moves it towards the end of the result. An OFFSET of zero leaves the position where it is, but causes the row to be re-fetched from the database. This operation is only available if the cursor is scrollable, that's if the statement has been created with CURSOR-TYPE-SCROLL-INSENSITIVE or CURSOR-TYPE-SCROLL-SENSITIVE.

OFFSET (input, optional)
A fullword binary variable containing the absolute or relative row number, dependent on the OPERATION parameter. If the OFFSET parameter is omitted, the offset defaults to zero. The OFFSET parameter should be specified for FETCH-ABSOLUTE and FETCH-RELATIVE. For all other operations it is ignored.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

GETCOLUMNINFO

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The GETCOLUMNINFO function allows an application to query the attributes of a column contained in the cursor after a statement has been executed. The application can then bind host variables to the column using the BINDCOLUMN function.

Related functions:
- “BINDCOLUMN” on page 301
- “GETNUMCOLUMNS” on page 373

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE            PIC S9(8) BINARY.
   01 STMT-HANDLE           PIC S9(8) BINARY.
   01 COL-INDEX             PIC S9(8) BINARY.
   01 NAME                  PIC X(nnn).
   01 NAME-LEN              PIC S9(8) BINARY.
   01 SQL-TYPE              PIC S9(8) BINARY.
   01 PRECISION             PIC S9(8) BINARY.
   01 SCALE                 PIC S9(8) BINARY.
   01 SIGNED                PIC S9(8) BINARY.
   01 NULLABLE              PIC S9(8) BINARY.
```

```
   01 RETCODE              PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-GETCOLUMNINFO ENV-HANDLE
        STMT-HANDLE COL-INDEX NAME NAME-LEN SQL-TYPE
        [PRECISION [SCALE [SIGNED [NULLABLE]]]]
        RETCODE.
```

FUNCTION (input)
A 16-byte character field containing GETCOLUMNINFO. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

STMT-HANDLE (input)
A fullword binary variable containing the statement handle.

COL-INDEX (input)
A fullword binary variable containing the number of the column to get info for. Column numbers start at 1.

NAME (output)
A character field that is set to the name of the column. The length of this field is specified using the NAME-LEN parameter. If the name of the column is larger than the length specified, it is silently truncated.

NAME-LEN (input)
A fullword binary variable containing the length of the NAME field.

SQL-TYPE (output)
A fullword binary variable that is set to the SQL-TYPE of the column. Please see the SQL-TYPE declarations in the copybook for possible values.

PRECISION (output, optional)
A fullword binary variable that is set to the precision of the column. This parameter is optional.

SCALE (output, optional)
A fullword binary variable that is set to the scale of the column. This parameter is optional. If it is specified, the PRECISION parameter must also be specified.

SIGNED (output, optional)
A fullword binary variable that is set to 0 if the column is unsigned, of 1 if the statement parameter is signed. This parameter is optional. If it is specified, the PRECISION and SCALE parameters must also be specified.

NULLABLE (output, optional)
A fullword binary variable that is set to 0 if the column can not contain NULL, of 1 if the statement parameter can contain NULL. This parameter is optional. If it is specified, the PRECISION, SCALE and SIGNED parameters must also be specified.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

GETCONNATTR

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The GETCONNATTR function allows an application to retrieve an attribute on the connection level.

Related function: “SETCONNATTR” on page 385.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE           PIC S9(8) BINARY.
   01 CON-HANDLE           PIC S9(8) BINARY.
   01 ATTR                 PIC S9(8) BINARY.
   01 VALUE                PIC S9(8) BINARY.
   01 VALUE-LEN            PIC S9(8) BINARY.
   01 RETCODE              PIC S9(8) BINARY.
PROCEDURE
  CALL 'IESDBCLI' USING FUNC-GETCONNATTR ENV-HANDLE CON-HANDLE
       ATTR VALUE VALUE-LEN RETCODE.
```

FUNCTION (input)
A 16-byte character field containing GETCONNATTR. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

ATTR (input)
A fullword binary variable containing the attribute ID of the attribute to set or get.

VALUE (input/output)
A field that receives the attribute value. Some attributes use textual values, therefore you must instead use a character field.

VALUE-LEN (input)
A fullword binary variable containing the length of the VALUE field. For attributes using a fullword binary value, the length should be set to 4.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The following connection attributes are defined:

CONNATTR-TRANSACTION-ISOLATION (1)
This attribute specifies the transaction isolation level used with this connection. The value is a fullword binary variable containing one of the following:

TRANSACTION-NONE (0):
No transactions are supported

TRANSACTION-READ-UNCOMMITTED (1):
Dirty reads, non-repeatable reads and phantom reads can occur.

TRANSACTION-READ-COMMITTED (2):
Dirty reads are prevented; non-repeatable reads and phantom reads can occur.

TRANSACTION-REPEATABLE-READ (4):
Dirty reads and non-repeatable reads are prevented; phantom reads can occur.

TRANSACTION-SERIALIZABLE (8):
Dirty reads, non-repeatable reads and phantom reads are prevented.

CONNATTR-READONLY (2)
This attribute specifies if the connection operates in read-only mode. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

CONNATTR-HOLDABILITY (3)
This attribute specifies if the default holdability mode used when creating statements using this connection. The value is a fullword binary variable containing one of the following:

HOLD-CURSORS-OVER-COMMIT (1):
Any open cursors are kept open when a `COMMIT` is performed.

CLOSE-CURSORS-AT-COMMIT (2):
Any open cursors are closed when a `COMMIT` is performed.

CONNATTR-CATALOG (4)
This attribute specifies the database catalog. The value is character field whose length is determined by the VALUE-LEN parameter.

CONNATTR-AUTO-COMMIT (5)
This attribute controls the auto-commit mode of this connection. If a connection is in auto-commit mode, all its SQL statements will be executed and committed as individual transactions. Otherwise, its SQL statements are grouped into transactions that are terminated by a call to the `COMMIT` or `ROLLBACK` function. By default, new connections are not in auto-commit mode. The value is a fullword binary variable containing 0 for FALSE (no auto-commit) and 1 for TRUE (auto commit enabled).

CONNATTR-STATNUMSTMTS (6)
This read only attribute can be used to retrieve the total number of statements that have been created on this connection. A `SETCONNATTR` call with this attribute will reset the counter to zero.

CONNATTR-STATNUMEXECS (7)
This read only attribute can be used to retrieve the total number of statement executions that have been performed (across all statements) on this connection. A `SETCONNATTR` call with this attribute will reset the counter to zero.

CONNATTR-STATNUMROWS (8)
This read only attribute can be used to retrieve the total number of rows that have been fetched (across all statements) on this connection. A `SETCONNATTR` call with this attribute will reset the counter to zero.

CONNATTR-SERVER-TRACE (9)
This attribute can be used to enable or disable the trace support on `DBCliServer` side. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

CONNATTR-SERVER-TRACE-LEVEL (10)
This attribute can be used to get or set the trace level of the trace support on `DBCliServer` side. The value is a fullword binary variable containing the trace level:

TRACELEVEL-FLOW (4)
Trace the main application flow

TRACELEVEL-NORMAL (5)
Trace most messages

TRACELEVEL-FINE (6)
Most extensive tracing

The following read-only connection attributes are defined to retrieve database meta information:

DBATTR-CATALOGSEPARATOR (16)
Retrieves the String that this database uses as the separator between a catalog and table name. The value is character field whose length is determined by the VALUE-LEN parameter.

DBATTR-CATALOGTERM (17)
Retrieves the database vendor's preferred term for "catalog". The value is character field whose length is determined by the VALUE-LEN parameter.

DBATTR-DATABASEPRODUCTNAME (18)
Retrieves the name of the database product to which this connection is connected to. The value is character field whose length is determined by the VALUE-LEN parameter.

DBATTR-DATABASEPRODUCTVERSION (19)
Retrieves the version number of the database product to which this connection is connected to. The value is character field whose length is determined by the VALUE-LEN parameter.

DBATTR-DRIVERNAME (20)
Retrieves the name of the JDBC driver through which this connection is connected to the database. The value is character field whose length is determined by the VALUE-LEN parameter.

DBATTR-DRIVERVERSION (21)
Retrieves the version number of the JDBC driver through which this connection is connected to the database. The value is character field whose length is determined by the VALUE-LEN parameter.

DBATTR-EXTRANAMECHARACTERS (22)
Retrieves all the "extra" characters that can be used in unquoted identifier names (those beyond a-z, A-Z, 0-9 and _). The value is character field whose length is determined by the VALUE-LEN parameter.

DBATTR-IDENTIFIERQUOTESTRING (23)
Retrieves the string used to quote SQL identifiers. This attribute returns a space " " if identifier quoting is not supported. The value is character field whose length is determined by the VALUE-LEN parameter. The value is character field whose length is determined by the VALUE-LEN parameter.

DBATTR-NUMERICFUNCTIONS (24)
Retrieves a comma-separated list of math functions available with the database to which this connection is connected to. These are the Open /Open CLI math function names used in the JDBC function escape clause. The value is character field whose length is determined by the VALUE-LEN parameter.

DBATTR-PROCEDURETERM (25)
Retrieves the database vendor's preferred term for "procedure". The value is character field whose length is determined by the VALUE-LEN parameter.

DBATTR-SQLKEYWORDS (26)
Retrieves a comma-separated list of all of the database's SQL keywords that are

NOT also SQL92 keywords. The value is character field whose length is determined by the VALUE-LEN parameter.

DBATTR-SCHEMATERM (27)
Retrieves the database vendor's preferred term for "schema". The value is character field whose length is determined by the VALUE-LEN parameter.

DBATTR-SEARCHSTRINGESCAPE (28)
Retrieves the string that can be used to escape wildcard characters. This is the string that can be used to escape '_' or '%' in the catalog search parameters that are a pattern (and therefore use one of the wildcard characters). The '_' character represents any single character; the '%' character represents any sequence of zero or more characters. The value is character field whose length is determined by the VALUE-LEN parameter.

DBATTR-STRINGFUNCTIONS (29)
Retrieves a comma-separated list of string functions available with the database. These are the Open Group CLI string function names used in the JDBC function escape clause. The value is character field whose length is determined by the VALUE-LEN parameter.

DBATTR-SYSTEMFUNCTIONS (30)
Retrieves a comma-separated list of system functions available with the database. These are the Open Group CLI system function names used in the JDBC function escape clause. The value is character field whose length is determined by the VALUE-LEN parameter.

DBATTR-TIMEDATEFUNCTIONS (31)
Retrieves a comma-separated list of the time and date functions available with this database. The value is character field whose length is determined by the VALUE-LEN parameter.

DBATTR-URL (32)
Retrieves the URL for this DBMS. The value is character field whose length is determined by the VALUE-LEN parameter.

DBATTR-USERNAME (33)
Retrieves the user name as known to this database. The value is character field whose length is determined by the VALUE-LEN parameter.

DBATTR-ALLPROCEDURESARECALLABLE (34)
Retrieves whether the current user can call all the procedures returned by the function DBPROCEDURES. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-ALLTABLESARESELECTABLE (35)
Retrieves whether the current user can use all the tables returned by the function DBTABLES in a SELECT statement. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-DATADEFCAUSESTRANSACTCOMMIT (36)
Retrieves whether a data definition statement within a transaction forces the transaction to commit. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-DATADEFIGNOREDINTRANSACT (37)
Retrieves whether this database ignores a data definition statement within a transaction. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-DELETESAREDETECTED (38)
Retrieves whether or not a visible row delete can be detected by the function

FETCH. If this attribute returns FALSE, it means that deleted rows are removed from the cursor. The value is a 12 byte area consisting of 3 fullword binary fields. The first fullword field is for the cursor type CURSOR-TYPE-FORWARD-ONLY (1003), the second for cursor type CURSOR-TYPE-SCROLL-INSENSITIVE (1004) and the third for cursor type CURSOR-TYPE-SCROLL-SENSITIVE(1005). The 3 fields contain 0 for FALSE and 1 for TRUE.

DBATTR-DOESMAXROWSIZEINCLUDEBLOBS (39)
Retrieves whether the value for the attribute DBATTR-MAXROWSIZE (145) includes the SQL data types LONGVARCHAR and LONGVARBINARY. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-INSERTSAREDETECTED (40)
Retrieves whether or not a visible row insert can be detected by the function FETCH. The value is a 12 byte area consisting of 3 fullword binary fields. The first fullword field is for the cursor type CURSOR-TYPE-FORWARD-ONLY (1003), the second for cursor type CURSOR-TYPE-SCROLL-INSENSITIVE (1004) and the third for cursor type CURSOR-TYPE-SCROLL-SENSITIVE(1005). The 3 fields contain 0 for FALSE and 1 for TRUE.

DBATTR-ISCATALOGATSTART (41)
Retrieves whether a catalog appears at the start of a fully qualified table name. If not, the catalog appears at the end. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-ISREADONLY (42)
Retrieves whether this database is in read-only mode. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-LOCATORSUPDATECOPY (43)
Indicates whether updates made to a LOB are made on a copy or directly to the LOB. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-NULLPLUSNONNULLISNULL (44)
Retrieves whether this database supports concatenations between NULL and non-NULL values being NULL. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-NULLSARESORTEDATEND (45)
Retrieves whether NULL values are sorted at the end regardless of sort order. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-NULLSARESORTEDATSTART (46)
Retrieves whether NULL values are sorted at the start regardless of sort order. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-NULLSARESORTEDHIGH (47)
Retrieves whether NULL values are sorted high. Sorted high means that NULL values sort higher than any other value in a domain. In an ascending order, if this attribute value is TRUE, NULL values will appear at the end. By contrast, the attribute DBATTR-NULLSARESORTEDATEND (45) indicates whether NULL values are sorted at the end regardless of sort order. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-NULLSARESORTEDLOW (48)
Retrieves whether NULL values are sorted low. Sorted low means that NULL values sort lower than any other value in a domain. In an ascending order, if this attribute value is TRUE, NULL values will appear at the beginning. By

contrast, the method DBATTR-NULLSARESORTEDATSTART (46) indicates whether NULL values are sorted at the beginning regardless of sort order. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-OTHERSDELETESAREVISIBLE (49)
Retrieves whether deletes made by others are visible. The value is a 12 byte area consisting of 3 fullword binary fields. The first fullword field is for the cursor type CURSOR-TYPE-FORWARD-ONLY (1003), the second for cursor type CURSOR-TYPE-SCROLL-INSENSITIVE (1004) and the third for cursor type CURSOR-TYPE-SCROLL-SENSITIVE(1005). The 3 fields contain 0 for FALSE and 1 for TRUE.

DBATTR-OTHERSINSERTSAREVISIBLE (50)
Retrieves whether inserts made by others are visible. The value is a 12 byte area consisting of 3 fullword binary fields. The first fullword field is for the cursor type CURSOR-TYPE-FORWARD-ONLY (1003), the second for cursor type CURSOR-TYPE-SCROLL-INSENSITIVE (1004) and the third for cursor type CURSOR-TYPE-SCROLL-SENSITIVE(1005). The 3 fields contain 0 for FALSE and 1 for TRUE.

DBATTR-OTHERSUPDATESAREVISIBLE (51)
Retrieves whether updates made by others are visible. The value is a 12 byte area consisting of 3 fullword binary fields. The first fullword field is for the cursor type CURSOR-TYPE-FORWARD-ONLY (1003), the second for cursor type CURSOR-TYPE-SCROLL-INSENSITIVE (1004) and the third for cursor type CURSOR-TYPE-SCROLL-SENSITIVE(1005). The 3 fields contain 0 for FALSE and 1 for TRUE.

DBATTR-OWNDELETESAREVISIBLE (52)
Retrieves whether a cursor's own deletes are visible. The value is a 12 byte area consisting of 3 fullword binary fields. The first fullword field is for the cursor type CURSOR-TYPE-FORWARD-ONLY (1003), the second for cursor type CURSOR-TYPE-SCROLL-INSENSITIVE (1004) and the third for cursor type CURSOR-TYPE-SCROLL-SENSITIVE(1005). The 3 fields contain 0 for FALSE and 1 for TRUE.

DBATTR-OWNINSERTSAREVISIBLE (53)
Retrieves whether a cursor's own inserts are visible. The value is a 12 byte area consisting of 3 fullword binary fields. The first fullword field is for the cursor type CURSOR-TYPE-FORWARD-ONLY (1003), the second for cursor type CURSOR-TYPE-SCROLL-INSENSITIVE (1004) and the third for cursor type CURSOR-TYPE-SCROLL-SENSITIVE(1005). The 3 fields contain 0 for FALSE and 1 for TRUE.

DBATTR-OWNUPDATESAREVISIBLE (54)
Retrieves whether a cursor's own updates are visible. The value is a 12 byte area consisting of 3 fullword binary fields. The first fullword field is for the cursor type CURSOR-TYPE-FORWARD-ONLY (1003), the second for cursor type CURSOR-TYPE-SCROLL-INSENSITIVE (1004) and the third for cursor type CURSOR-TYPE-SCROLL-SENSITIVE(1005). The 3 fields contain 0 for FALSE and 1 for TRUE.

DBATTR-STORESLOWERCASEIDENT (55)
Retrieves whether the database treats mixed case unquoted SQL identifiers as case insensitive and stores them in lower case. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-STORESLOWERCASEQUOTEDIDENT (56)
Retrieves whether this database treats mixed case quoted SQL identifiers as

case insensitive and stores them in lower case. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-STORESMIXEDCASEIDENT (57)
Retrieves whether the database treats mixed case unquoted SQL identifiers as case insensitive and stores them in mixed case. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-STORESMIXEDCASEQUOTEDIDENT (58)
Retrieves whether this database treats mixed case quoted SQL identifiers as case insensitive and stores them in mixed case. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-STORESUPPERCASEIDENT (59)
Retrieves whether the database treats mixed case unquoted SQL identifiers as case insensitive and stores them in upper case. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-STORESUPPERCASEQUOTEDIDENT (60)
Retrieves whether this database treats mixed case quoted SQL identifiers as case insensitive and stores them in upper case. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPANSI92ENTRYLEVELSQL (61)
Retrieves whether this database supports the ANSI92 entry level SQL grammar. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPANSI92FULLSQL (62)
Retrieves whether this database supports the ANSI92 full SQL grammar. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPANSI92INTERMSQL (63)
Retrieves whether this database supports the ANSI92 intermediate SQL grammar. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPALTTABLEWITHADDCOL (64)
Retrieves whether this database supports ALTER TABLE with add column. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPALTTABLEWITHDROPCOL (65)
Retrieves whether this database supports ALTER TABLE with drop column. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPBATCHUPDATES (66)
Retrieves whether this database supports batch updates. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPCATSINDATAMANIPULATION (67)
Retrieves whether a catalog name can be used in a data manipulation statement. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPCATSININDEXDEFINITIONS (68)
Retrieves whether a catalog name can be used in an index definition statement. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPCATSINPRIVILEGEDEFS (69)
Retrieves whether a catalog name can be used in a privilege definition statement. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPCATSINPROCEDURECALLS (70)
Retrieves whether a catalog name can be used in a procedure call statement. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPCATSINTABLEDEFINITIONS (71)
Retrieves whether a catalog name can be used in a table definition statement. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPCOLUMNALIASING (72)
Retrieves whether this database supports column aliasing. If so, the SQL AS clause can be used to provide names for computed columns or to provide alias names for columns as required. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPCORESQLGRAMMAR (73)
Retrieves whether this database supports the ODBC Core SQL grammar. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPCORRELATEDSUBQUERIES (74)
Retrieves whether this database supports correlated subqueries. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPDATADEFANDDATAMANTRANS (75)
Retrieves whether this database supports both data definition and data manipulation statements within a transaction. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPDATAMANTRANSACTIONSONLY (76)
Retrieves whether this database supports only data manipulation statements within a transaction. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPDIFFTABLECORRNAMES (77)
Retrieves whether, when table correlation names are supported, they are restricted to being different from the names of the tables. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPEXPRESSIONSINORDERBY (78)
Retrieves whether this database supports expressions in ORDER BY lists. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPEXTENDEDSQLGRAMMAR (79)
Retrieves whether this database supports the ODBC Extended SQL grammar. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPFULLOUTERJOINS (80)
Retrieves whether this database supports full nested outer joins. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPGETGENERATEDKEYS (81)
Retrieves whether auto-generated keys can be retrieved after a statement has been executed. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPGROUPBY (82)
Retrieves whether this database supports some form of GROUP BY clause. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPGROUPBYBEYONDSELECT (83)
Retrieves whether this database supports using columns not included in the SELECT statement in a GROUP BY clause provided that all of the columns in the SELECT statement are included in the GROUP BY clause. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPGROUPBYUNRELATED (84)
Retrieves whether this database supports using a column that is not in the SELECT statement in a GROUP BY clause. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPINTEGRITYENHANCEMENTFAC (85)
Retrieves whether this database supports the SQL Integrity Enhancement Facility. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPLIKEESCAPECLAUSE (86)
Retrieves whether this database supports specifying a LIKE escape clause. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPLIMITEDOUTERJOINS (87)
Retrieves whether this database provides limited support for outer joins. (This attribute will be TRUE if the attribute DBATTR-SUPFULLOUTERJOINS (80) is TRUE). The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPMINIMUMSQLGRAMMAR (88)
Retrieves whether this database supports the ODBC Minimum SQL grammar. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPMIXEDCASEIDENTIFIERS (89)
Retrieves whether this database treats mixed case unquoted SQL identifiers as case sensitive and as a result stores them in mixed case. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPMIXEDCASEQUOTEDIDENT (90)
Retrieves whether this database treats mixed case quoted SQL identifiers as case sensitive and as a result stores them in mixed case. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPMULTIPLEOPENRESULTS (91)
Retrieves whether it is possible to have multiple cursors returned from a stored procedure call simultaneously. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPMULTIPLERESULTSETS (92)
Retrieves whether this database supports getting multiple cursors from a single call to the EXECUTE function. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPMULTIPLETRANSACTIONS (93)
Retrieves whether this database allows having multiple transactions open at once (on different connections). The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPNAMEDPARAMETERS (94)
Retrieves whether this database supports named parameters to callable statements. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPNONNULLABLECOLUMNS (95)
Retrieves whether columns in this database may be defined as non-nullable. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPOPENCURACROSSCOMMIT (96)
Retrieves whether this database supports keeping cursors open across commits. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPOPENCURACROSSROLLBACK (97)
Retrieves whether this database supports keeping cursors open across rollbacks. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPOPENSTMTSACROSSCOMMIT (98)
Retrieves whether this database supports keeping statements open across commits. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPOPENSTMTSACROSSROLLBACK (99)
Retrieves whether this database supports keeping statements open across rollbacks. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPORDERBYUNRELATED (100)
Retrieves whether this database supports using a column that is not in the SELECT statement in an ORDER BY clause. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPOUTERJOINS (101)
Retrieves whether this database supports some form of outer join. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPPOSITIONEDDELETE (102)
Retrieves whether this database supports positioned DELETE statements. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPPOSITIONEDUPDATE (103)
Retrieves whether this database supports positioned UPDATE statements. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPRESULTSETTYPE (104)
Retrieves whether this database supports the given result set type. The value is a 12 byte area consisting of 3 fullword binary fields. The first fullword field is for the cursor type CURSOR-TYPE-FORWARD-ONLY (1003), the second for cursor type CURSOR-TYPE-SCROLL-INSENSITIVE (1004), and the third for cursor type CURSOR-TYPE-SCROLL-SENSITIVE(1005). The 3 fields contain 0 for FALSE and 1 for TRUE.

DBATTR-SUPSAVEPOINTS (105)
Retrieves whether this database supports savepoints. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPSCHEMASINDATAMANIPUL (106)
Retrieves whether a schema name can be used in a data manipulation statement. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPSCHEMASININDEXDEFS (107)
Retrieves whether a schema name can be used in an index definition statement. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPSCHEMASINPRIVILEGEDEFS (108)
Retrieves whether a schema name can be used in a privilege definition statement. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPSCHEMASINPROCEDURECALLS (109)
Retrieves whether a schema name can be used in a procedure call statement. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPSCHEMASINTABLEDEFS (110)
Retrieves whether a schema name can be used in a table definition statement. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPSELECTFORUPDATE (111)
Retrieves whether this database supports SELECT FOR UPDATE statements. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPSTMTPOOLING (112)
Retrieves whether this database supports statement pooling. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPSTOREDPROCEDURES (113)
Retrieves whether this database supports stored procedure calls that use the stored procedure escape syntax. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPSUBQUERIESINCOMPARISONS (114)
Retrieves whether this database supports subqueries in comparison expressions. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPSUBQUERIESINEXISTS (115)
Retrieves whether this database supports subqueries in EXISTS expressions. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPSUBQUERIESININS (116)
Retrieves whether this database supports subqueries in IN statements. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPSUBQUERIESINQUANTIFIEDS (117)
Retrieves whether this database supports subqueries in quantified expressions. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPTABLECORRELATIONNAMES (118)
Retrieves whether this database supports table correlation names. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPTRANSACTIONS (119)
Retrieves whether this database supports transactions. If not, calling the COMMIT function is a noop, and the isolation level is TRANSACTION-NONE (0). The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPUNION (120)
Retrieves whether this database supports SQL UNION. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-SUPUNIONALL (121)
Retrieves whether this database supports SQL UNION ALL. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-UPDATESAREDETECTED (122)
Retrieves whether or not a visible row update can be detected by the FETCH function. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE

DBATTR-USESLOCALFILEPERTABLE (123)
Retrieves whether this database uses a file for each table. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-USESLOCALFILES (124)
Retrieves whether this database stores tables in a local file. The value is a fullword binary variable containing 0 for FALSE and 1 for TRUE.

DBATTR-DATABASEMAJORVERSION (125)
Retrieves the major version number of the underlying database. The value is a fullword binary variable.

DBATTR-DATABASEMINORVERSION (126)
Retrieves the minor version number of the underlying database. The value is a fullword binary variable.

DBATTR-DEFAULTTRANSACTIONISOLATION (127)
Retrieves this database's default transaction isolation level. The value is a fullword binary variable containing TRANSACTION-NONE(0), TRANSACTION-READ-UNCOMMITTED (1), TRANSACTION-READ-COMMITTED (2), TRANSACTION-REPEATABLE-READ (4) or TRANSACTION-SERIALIZABLE (8).

DBATTR-DRIVERMAJORVERSION (128)
Retrieves this JDBC driver's major version number. The value is a fullword binary variable.

DBATTR-DRIVERMINORVERSION (129)
Retrieves this JDBC driver's minor version number. The value is a fullword binary variable.

DBATTR-JDBCMAJORVERSION (130)
Retrieves the major JDBC version number for this driver. The value is a fullword binary variable.

DBATTR-JDBCMINORVERSION (131)
Retrieves the minor JDBC version number for this driver. The value is a fullword binary variable.

DBATTR-MAXBINARYLITERALLENGTH (132)
Retrieves the maximum number of hex characters this database allows in an inline binary literal. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-MAXCATALOGNAMELENGTH (133)
Retrieves the maximum number of characters that this database allows in a catalog name. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-MAXCHARLITERALLENGTH (134)
Retrieves the maximum number of characters this database allows for a character literal. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-MAXCOLUMNNAMELENGTH (135)
Retrieves the maximum number of characters this database allows for a column name. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-MAXCOLUMNSINGROUPBY (136)
Retrieves the maximum number of columns this database allows in a GROUP BY clause. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-MAXCOLUMNSININDEX (137)
Retrieves the maximum number of columns this database allows in an index. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-MAXCOLUMNSINORDERBY (138)
Retrieves the maximum number of columns this database allows in an ORDER BY clause. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-MAXCOLUMNSINSELECT (139)
Retrieves the maximum number of columns this database allows in a SELECT list. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-MAXCOLUMNSINTABLE (140)
Retrieves the maximum number of columns this database allows in a table. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-MAXCONNECTIONS (141)
Retrieves the maximum number of concurrent connections to this database that are possible. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-MAXCURSORNAMELENGTH (142)
Retrieves the maximum number of characters that this database allows in a cursor name. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-MAXINDEXLENGTH (143)
Retrieves the maximum number of bytes this database allows for an index, including all of the parts of the index. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-MAXPROCEDURENAMELENGTH (144)
Retrieves the maximum number of characters that this database allows in a procedure name. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-MAXROWSIZE (145)
Retrieves the maximum number of bytes this database allows in a single row. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-MAXSCHEMANAMELENGTH (146)
Retrieves the maximum number of characters that this database allows in a schema name. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-MAXSTMTLENGTH (147)
Retrieves the maximum number of characters this database allows in an SQL statement. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-MAXSTMTS (148)
Retrieves the maximum number of active statements to this database that can be open at the same time. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-MAXTABLENAMELENGTH (149)
Retrieves the maximum number of characters this database allows in a table name. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-MAXTABLESINSELECT (150)
Retrieves the maximum number of tables this database allows in a SELECT statement. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-MAXUSERNAMELENGTH (151)
Retrieves the maximum number of characters this database allows in a user name. A value of zero means that there is no limit or the limit is not known. The value is a fullword binary variable.

DBATTR-RESULTSETHOLDABILITY (152)
Retrieves the default holdability of a cursor. The value is a fullword binary variable containing either HOLD-CURSORS-OVER-COMMIT (1) or CLOSE-CURSORS-AT-COMMIT (2).

DBATTR-SQLSTATETYPE (153)
Indicates whether the SQLSTATE returned by GETLASTERROR is X/Open (now known as Open Group) SQL CLI or SQL99. The value is a fullword binary variable containing either 1 for X/Open or 2 for SQL99.

DBATTR-SUPRESULTSETCONCURRENCY (154)
Retrieves whether this database supports the concurrency type in combination with the cursor type. The value is a 24 byte area consisting of 6 fullword binary fields. The first fullword field is for the cursor type CURSOR-TYPE-FORWARD-ONLY (1003) and CURSOR-CONCUR-READ-ONLY (1007), the second for cursor type CURSOR-TYPE-SCROLL-INSENSITIVE (1004) and CURSOR-CONCUR-READ-ONLY (1007), and the third for cursor type CURSOR-TYPE-SCROLL-SENSITIVE(1005) and CURSOR-CONCUR-READ-ONLY (1007). The fourth fullword field is for the cursor type CURSOR-TYPE-FORWARD-ONLY (1003) and CURSOR-CONCUR-UPDATABLE (1008), the fifth for cursor type CURSOR-TYPE-SCROLL-INSENSITIVE (1004) and CURSOR-CONCUR- UPDATABLE (1008), and the sixth for cursor type CURSOR-TYPE-SCROLL-SENSITIVE(1005) and CURSOR-CONCUR-UPDATABLE (1008). The 6 fields contain 0 for FALSE and 1 for TRUE.

DBATTR-SUPRESULTSETHOLDABILITY (155)
Retrieves whether this database supports the cursor holdabilities. The value is an 8 byte area consisting of 2 fullword binary fields. The first fullword field is for the cursor holdability HOLD-CURSORS-OVER-COMMIT (1), the second for cursor type CLOSE-CURSORS-AT-COMMIT (2). The 2 fields contain 0 for FALSE and 1 for TRUE.

DBATTR-SUPTRANSACTISOLATIONLEVEL (156)
Retrieves whether this database supports the transaction isolation levels. The value is an 20 byte area consisting of 5 fullword binary fields. The first fullword field is for the isolation level TRANSACTION-NONE (0), the second for TRANSACTION-READ-UNCOMMITTED (1), the third for TRANSACTION-READ-COMMITTED (2), the fourth for TRANSACTION-REPEATABLE-READ (4) and the fifth for TRANSACTION-SERIALIZABLE (8), The 5 fields contain 0 for FALSE and 1 for TRUE.

GETENVATTR

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The GETENVATTR function allows an application to retrieve an attribute on the environment level. In general, environment attributes should be set before a connection is established using the CONNECT function.

Related function: "SETENVATTR" on page 386.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 ATTR               PIC S9(8) BINARY.
   01 VALUE              PIC S9(8) BINARY.
   01 VALUE-LEN          PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-GETENVATTR ENV-HANDLE
        ATTR VALUE VALUE-LEN RETCODE.
```

FUNCTION (input)
A 16-byte character field containing GETENVATTR. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

ATTR (input)
A fullword binary variable containing the attribute ID of the attribute to set or get.

VALUE (input/output)
A field that receives the attribute value. Some attributes use textual values, therefore you must instead use a character field.

VALUE-LEN (input)
A fullword binary variable containing the length of the VALUE field. For attributes using a fullword binary value, the length should be set to 4.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The following environment attributes are defined:

ENVATTR-TRACE (1)
Allows enabling or disabling the DBCLI internal trace. Tracing can be enabled or disabled dynamically. The attribute is effective immediately. The value is a fullword binary variable containing one of the following:

TRACE-OFF (0):
Tracing is disabled.

TRACE-SYSLST (1):
Tracing to SYSLST is enabled

TRACE-SYSLOG (2):
Tracing to SYSLOG is enabled.

TRACE-BOTH (3):
Trace to both SYSLST and SYSLOG is enabled.

ENVATTR-RECV-BUFFER-SIZE (2)
This attribute controls the size of the internal receive buffer used when creating new connections. The default receive buffer size is 32KB. The value is a fullword binary variable containing the size of the buffer in bytes. If this attribute is set to a value of zero, the receive buffer size is set to the default size (32K). The minimum receive buffer size is 64. Setting it to anything less than 64 will set it to 64. Changing this attribute does not affect already existing connections. For SSL type connections, the minimum receive buffer size is 32K.

ENVATTR-SEND-BUFFER-SIZE (3)
This attribute controls the size of the internal send buffer used when creating new connections. The default send buffer size is 32KB. The value is a fullword binary variable containing the size of the buffer in bytes. If this attribute is set to a value of zero, the send buffer size is set to the default size (32K). The minimum send buffer size is 64. Setting it to anything less than 64 will set it to 64. Changing this attribute does not affect already existing connections. For SSL type connections, the minimum send buffer size is 32K.

ENVATTR-DEFAULT-PORT (4)
This attribute specifies the default port number that is used when a port number of 0 is specified with the CONNECT function. The value is a fullword binary variable. The initial default port number is 16178. Changing this attribute does not affect already existing connections.

ENVATTR-CODEPAGE (5)
This attribute specifies the default EBCDIC code page used to convert textual data from EBCDIC to ASCII and vice versa. The value is a character field of up to 16 characters. The default code page is Cp1047. Changing this attribute does not affect already existing connections.

ENVATTR-STATNUMCONS (6)
This read only attribute can be used to retrieve the total number of connections that have been opened since the environment has been created. A `SETENVATTR` call with this attribute will reset the counter to zero.

ENVATTR-STATNUMSTMTS (7)
This read only attribute can be used to retrieve the total number of statements that have been opened since the environment has been created. A `SETENVATTR` call with this attribute will reset the counter to zero.

ENVATTR-STATNUMEXECS (8)
This read only attribute can be used to retrieve the total number of statement

executions that have been performed (across all statements and connections) since the environment has been created. A SETENVATTR call with this attribute will reset the counter to zero.

ENVATTR-STATNUMROWS (9)
This read only attribute can be used to retrieve the total number of rows that have been fetched (across all statements and connections) since the environment has been created. A SETENVATTR call with this attribute will reset the counter to zero.

ENVATTR-USE-CONNPOOL (10)
This attribute can be used to enable the use of the connection pool for an application. Per default, the connection pool is not used. To enable the use of the connection pool, an application must set this attribute to TRUE (1) via the SETENVATTR function.

ENVATTR-CONNPOOL-TIMEOUT (11)
This attribute can be used to specify the length of time (in seconds) that a connection, put into the connection pool, will be kept. If the connection has not been reused during this time, it will be closed. The default timeout is 600 seconds (10 minutes). An application can change the timeout by setting the ENVATTR-CONNPOOL-TIMEOUT attribute via the SETENVATTR function.

ENVATTR-CONNPOOL-MAXCONS (12)
This attribute can be used to specify the maximum number of identical connections (that is, those with the same hostname/IP address, port, database name, user-ID, and password) that can be stored in the connection pool. If a connection is put back into the connection pool and there are already more than the specified maximum of the identical connections in the connection pool, the connection will be closed. The default number is 16. An application can change the maximum number by setting the ENVATTR-CONNPOOL-MAXCONS attribute via the SETENVATTR function.

GETLASTERROR

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The GETLASTERROR function allows an application to retrieve error details about the last call. The error details are available for most functions and error situations. They include a textual error message, as well as an SQLCODE and SQLSTATE if available. The GETERRORMSG functions returns a return code, but does not set the SQLCODE, SQLSTATE or error message.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE       PIC S9(8) BINARY.
   01 SQLCODE          PIC S9(8) BINARY.
   01 SQLSTATE         PIC X(5).
   01 ERRMSG           PIC X(nnn).
   01 ERRMSG-LEN       PIC S9(8) BINARY.
   01 RETCODE          PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-GETLASTERROR ENV-HANDLE
        SQLCODE SQLSTATE [ERRMSG ERRMSG-LEN]
        RETCODE.
```

FUNCTION (input)
A 16-byte character field containing GETLASTERROR. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

SQLCODE (output)
A fullword binary variable that is set to the SQLCODE from the last call or 0 if no SQLCODE is available.

SQLSTATE (output)
A 5-byte character field that is set to the SQLSTATE from the last call or blank if no SQLSTATE is available.

ERRMSG (output, optional)
A character field that is set to the textual error message from the last call or blank if no error message is available. If the ERRMSG parameter is specified, the ERRMSG-LEN parameter must also be specified. If the error message is larger than the ERRMSG field (as specified by ERRMSG-LEN) then the message is truncated.

ERRMSG-LEN (input, optional)
A fullword binary variable containing the length of the ERRMSG field. If the ERRMSG-LEN parameter is specified, the ERRMSG parameter must also be specified.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

GETMORERESULTS

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The GETMORERESULTS function checks if more results are available, and moves the cursor to the next result, if available. If no more results are available, the GETMORERESULTS function returns ENOMOREDATA.

- A result can either be another cursor, or an update count.
- A call to the GETMORERESULTS function will implicitly close any open cursor.
- If the statement was a stored procedure call, the host variables bound to output parameters will be set to the value set by the stored procedure.
- Output parameters are only set when no more results are available. Thus, the GETMORERESULTS function should be called until ENOMOREDATA is returned.
- If the statement was an update, the update count is provided with this function, or can be retrieved at a later time using the GETUPDATECOUNT function.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE           PIC S9(8) BINARY.
   01 STMT-HANDLE          PIC S9(8) BINARY.
   01 UPDATE-COUNT         PIC S9(8) BINARY.
   01 RETCODE              PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-GETMORERESULTS ENV-HANDLE
        STMT-HANDLE [UPDATE-COUNT] RETCODE.
```

FUNCTION (input)
A 16-byte character field containing GETMORERESULTS. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

STMT-HANDLE (input)
A fullword binary variable containing the statement handle.

UPDATE-COUNT (ouput, optional)
A fullword binary variable that is set to the number of rows updated by the statement execution or -1 if no update count is available.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

GETNUMCOLUMNS

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The GETNUMCOLUMNS function allows an application to query the number of columns contained in the cursor after a statement has been executed. The application can then bind host variables the columns using the BINDCOLUMN function.

Related functions:
- "BINDCOLUMN" on page 301
- "GETCOLUMNINFO" on page 353

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE           PIC S9(8) BINARY.
   01 STMT-HANDLE          PIC S9(8) BINARY.
   01 NUM-COLUMNS          PIC S9(8) BINARY.
   01 RETCODE              PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-GETNUMCOLUMNS ENV-HANDLE
        STMT-HANDLE NUM-COLUMNS RETCODE.
```

FUNCTION (input)
A 16-byte character field containing GETNUMCOLUMNS. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

STMT-HANDLE (input)
A fullword binary variable containing the statement handle.

NUM-COLUMNS (ouput)
A fullword binary variable that is set to the number of columns contained in the cursor after a statement has been executed. If no cursor is available, an error is returned.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

GETNUMPARAMETERS

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The GETNUMPARAMETERS function allows an application to query the number of parameters contained in a statement. The application can then bind host variables to the parameters using the BINDPARAMETER function.

Related functions:

- "BINDPARAMETER" on page 304
- "GETPARAMETERINFO"

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE            PIC S9(8) BINARY.
   01 STMT-HANDLE           PIC S9(8) BINARY.
   01 NUM-PARAMS            PIC S9(8) BINARY.
   01 RETCODE               PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-GETNUMPARAMETERS ENV-HANDLE
        STMT-HANDLE NUM-PARAMS RETCODE.
```

FUNCTION (input)
: A 16-byte character field containing GETNUMPARAMETERS. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
: A fullword binary variable containing the environment handle.

STMT-HANDLE (input)
: A fullword binary variable containing the statement handle.

NUM-PARAMS (output)
: A fullword binary variable that is set to the number of parameters of this statement.

RETCODE (output)
: A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

GETPARAMETERINFO

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The GETPARAMETERINFO function allows an application to query the attributes of a parameter used in a statement. The application can then bind host variables to the parameter using the BINDPARAMETER function.

Related functions:

- "BINDPARAMETER" on page 304
- "GETNUMPARAMETERS"

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 STMT-HANDLE        PIC S9(8) BINARY.
   01 PARAM-INDEX        PIC S9(8) BINARY.
   01 SQL-TYPE           PIC S9(8) BINARY.
   01 PRECISION          PIC S9(8) BINARY.
   01 SCALE              PIC S9(8) BINARY.
   01 SIGNED             PIC S9(8) BINARY.
```

```
   01 NULLABLE           PIC S9(8) BINARY.
   01 INOUTMODE          PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-GETPARAMETERINFO ENV-HANDLE
        STMT-HANDLE PARM-INDEX SQL-TYPE [PRECISION [SCALE [SIGNED
        [NULLABLE [INOUTMODE]]]]] RETCODE.
```

FUNCTION (input)
A 16-byte character field containing GETPARAMETERINFO. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

STMT-HANDLE (input)
A fullword binary variable containing the statement handle.

PARAM-INDEX (input)
A fullword binary variable containing the number of the statement parameter to get info for. Parameter numbers start at 1.

SQL-TYPE (output)
A fullword binary variable that is set to the SQL-TYPE of the statement parameter. Please see the SQL-TYPE declarations in the copybook for possible values.

PRECISION (output, optional)
A fullword binary variable that is set to the precision of the statement parameter. This parameter is optional.

SCALE (output, optional)
A fullword binary variable that is set to the scale of the statement parameter. This parameter is optional. If it is specified, the PRECISION parameter must also be specified.

SIGNED (output, optional)
A fullword binary variable that is set to 0 if the statement parameter is unsigned, of 1 if the statement parameter is signed. This parameter is optional. If it is specified, the PRECISION and SCALE parameters must also be specified.

NULLABLE (output, optional)
A fullword binary variable that is set to 0 if the statement parameter can not contain NULL, of 1 if the statement parameter can contain NULL. This parameter is optional. If it is specified, the PRECISION, SCALE and SIGNED parameters must also be specified.

INOUTMODE (output, optional)
A fullword binary variable that is set to the input/output mode of the statement parameter. Possible values are INOUT-MODE-UNKNOWN (0), INOUT-MODE-IN (1), INOUT-MODE-OUT (2), INOUT-MODE-INOUT (4). This parameter is optional. If it is specified, the PRECISION, SCALE, SIGNED and NULLABLE parameters must also be specified.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

GETROWNUMBER

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The GETROWNUMBER function allows an application to query the current row number where the cursor is currently positioned. If no open cursor is available, the function returns an error. An alternative method to obtain the current row number is to bind a fullword binary host variable to column number 0 using the BINDCOLUMN function. With this, the host variable will be set to the current row number whenever the FETCH function is called.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 STMT-HANDLE        PIC S9(8) BINARY.
   01 ROW-NUMBER         PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-GETROWNUMBER ENV-HANDLE
        STMT-HANDLE ROW-NUMBER RETCODE.
```

FUNCTION (input)
A 16-byte character field containing GETROWNUMBER. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

STMT-HANDLE (input)
A fullword binary variable containing the statement handle.

ROW-NUMBER (ouput)
A fullword binary variable that is set to the current row number where the cursor is currently positioned. If no cursor is available, an error is returned.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

GETSTMTATTR

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The GETSTMTATTR function allows an application to retrieve an attribute on the statement level.

Related function: "SETSTMTATTR" on page 389.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE            PIC S9(8) BINARY.
   01 STMT-HANDLE           PIC S9(8) BINARY.
   01 ATTR                  PIC S9(8) BINARY.
   01 VALUE                 PIC S9(8) BINARY.
   01 VALUE-LEN             PIC S9(8) BINARY.
   01 RETCODE               PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-GETSTMTATTR ENV-HANDLE STMT-HANDLE
        ATTR VALUE VALUE-LEN RETCODE.
```

FUNCTION (input)
A 16-byte character field containing GETSTMTATTR. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

STMT-HANDLE (input)
A fullword binary variable containing the statement handle.

ATTR (input)
A fullword binary variable containing the attribute ID of the attribute to set or get.

VALUE (input/output)
A field that receives the attribute value. Some attributes use textual values, therefore you must instead use a character field.

VALUE-LEN (input)
A fullword binary variable containing the length of the VALUE field. For attributes using a fullword binary value, the length should be set to 4.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

The following statement attributes are defined:

STMTATTR-PREFETCHROWS (1)
This attribute specifies the number of rows that are prefetched when a FETCH function is performed. The value is a fullword binary variable containing the number of rows to prefetch. The default is 128 rows. Higher values may be beneficial for performance, but require more memory. Setting this attribute to zero will set it to the default of 128 rows.

STMTATTR-FETCHSIZE (2)
This attribute gives the JDBC driver a hint as to the number of rows that should be fetched from the database when more rows are needed. If the value specified is zero, the hint is ignored. The default value is zero.

STMTATTR-QUERYTIMEOUT (3)
This attribute specifies the number of seconds the JDBC driver will wait for a statement to execute. If the limit is exceeded, an SQL error is produced. Zero means there is no limit. The default value is zero.

STMTATTR-MAXFIELDSIZE (4)
This attribute specifies the limit for the maximum number of bytes in a cursor column storing character or binary values to the given number of bytes. This limit applies only to BINARY, VARBINARY, LONGVARBINARY, CHAR, VARCHAR, and LONGVARCHAR fields. If the limit is exceeded, the excess data is silently discarded. For maximum portability, use values greater than 256. Zero means there is no limit. The default value is zero.

STMTATTR-MAXROWS (5)
This attribute specifies the limit for the maximum number of rows that any cursor can contain. If the limit is exceeded, the excess rows are silently dropped. Zero means there is no limit. The default value is zero.

STMTATTR-TYPE (6)
This attribute is read only. It contains the cursor type. The cursor type controls if the cursor is forward-only, or if it can be used for scrolling. The following values can be used:

CURSOR-TYPE-FORWARD-ONLY (1003):
The cursor can move only forward.

CURSOR-TYPE-SCROLL-INSENSITIVE (1004):
The cursor is scrollable but generally not sensitive to changes made by others.

CURSOR-TYPE-SCROLL-SENSITIVE (1005):
The cursor is scrollable and generally sensitive to changes made by others.

STMTATTR-CONCURRENCY (7)
This attribute is read only. It contains concurrency mode. The concurrency mode controls if the cursor is read-only, or if it can be updated. The following values can be used:

CURSOR-CONCUR-READ-ONLY (1007):
The cursor is read-only.

CURSOR-CONCUR-UPDATABLE (1008):
The cursor is updatable.

STMTATTR-HOLDABILITY (8)
This attribute is read only. It contains the holdability mode. The value is a fullword binary variable containing one of the following:

HOLD-CURSORS-OVER-COMMIT (1):
Any open cursors are kept open when a COMMIT is performed.

CLOSE-CURSORS-AT-COMMIT (2):
Any open cursors are closed when a COMMIT is performed.

STMTATTR-STATNUMEXECS (9)
This read only attribute can be used to retrieve the total number of times this statement has been executed.

STMTATTR-STATNUMROWS (10)
This read only attribute can be used to retrieve the total number of rows that have been fetched using this statement.

STMTATTR-NOBINDCHECK (11)
This attribute can be used to turn "off" the checking of whether all input parameters are bound when a statement is executed through the EXECUTE function. Per default this attribute is FALSE (0) which means that bind checking *is* performed. To turn bind checking "off", set this attribute to TRUE (1).

GETUPDATECOUNT

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The GETUPDATECOUNT function retrieves the update count value from the last statement execution.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE           PIC S9(8) BINARY.
   01 STMT-HANDLE          PIC S9(8) BINARY.
   01 UPDATE-COUNT         PIC S9(8) BINARY.
   01 RETCODE              PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-GETUPDATECOUNT ENV-HANDLE
        STMT-HANDLE UPDATE-COUNT RETCODE.
```

FUNCTION (input)
A 16-byte character field containing GETUPDATECOUNT. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

STMT-HANDLE (input)
A fullword binary variable containing the statement handle.

UPDATE-COUNT (ouput)
A fullword binary variable that is set to the number of rows updated by the statement execution or -1 if no update count is available.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

INITENV

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The INITENV function must be the very first call to the DBCLI API. It initializes the environment, including the EZA TCP/IP interface. The INITENV call allocates the environment handle and sets the ENV-HANDLE parameter. The application must pass the environment handle to all subsequent functions.

Note: Before using this function, ensure you have read "Programming Restrictions and Requirements" on page 296 (especially the note about using the DBCLI API in CICS transactions).

Related function: "TERMENV" on page 389.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE              PIC S9(8) BINARY.
   01 IDENT.
       02 TCPNAME             PIC X(8).
       02 ADSNAME             PIC X(8).
   01 RETCODE                 PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-INITENV ENV-HANDLE
        [TCPNAME [ADSNAME]] RETCODE.
```

FUNCTION (input)
A 16-byte character field containing INITENV. The field is left justified and padded on the right with blanks.

ENV-HANDLE (output)
A fullword binary variable containing the environment handle. The ENV-HANDLE value set by the INITENV call must be specified with every subsequent call to other DBCLI functions.

TCPNAME (input, optional)
This optional parameter can be used to select the local TCP/IP stack used with this application. This 8-byte parameter can be set to "SOCKETnn" or just to "nn" (left- or right-adjusted, padded with 6 blanks). The value "nn" determines the ID of the selected TCP/IP stack as it is specified with the ID parameter in the TCP/IP startup JCL. If TCPNAME is not specified, the ID from the // OPTION SYSPARM='nn' statement is used. The ID defaults to '00' if none of above are specified.

ADSNAME (input, optional)
The optional parameter can be used to specify the name of the TCP/IP Interface Routine used by the EZA processing environment. If nothing is

specified here, the IBM-supplied TCP/IP Interface Routine EZASOH99 is used. Please note that this specification can be overwritten with the following JCL statement: // SETPARM EZA$PHA='routine-name'. The ADSNAME parameter can only be specified if the TCPNAME parameter is also specified.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

INITSSL

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

If you plan to use SSL connections, you must initialize the SSL environment before calling the CONNECTSSL function.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE       PIC S9(8) BINARY.
   01 SECTYPE          PIC X(8) VALUE IS 'SSL30'.
   01 KEYRING          PIC X(nnn) VALUE IS 'CRYPTO.KEYRING'.
   01 KEYRING-LEN      PIC S9(8) BINARY VALUE IS 14.
   01 SESS-TIMOUT      PIC S9(8) BINARY VALUE IS 86400.
   01 RETCODE          PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-INITSSL ENV-HANDLE
        SECTYPE KEYRING KEYRING-LEN [SESS-TIMEOUT] RETCODE.
```

FUNCTION (input)
A 16-byte character field containing INITSSL. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

SECTYPE (input)
This parameter identifies the minimum acceptable security protocol. The value must be entered in upper case characters and padded with blanks. It can have the following values: "SSL30" for SSL Version 3.0, "TLS31" for TLS Version 1.0.

KEYRING (input)
The parameter specifies the keyring to read the SSL keys and certificates from. It usually specifies a z/VSE library and sublibrary (*lib.sublib*) where the key material is stored in.

KEYRING-LEN (input)
A fullword binary variable containing the length of the KEYRING field.

SESS-TIMEOUT (input, optional)
A fullword binary variable containing the SSL session timeout in seconds. If this parameter is not specified, the default timeout of 86400 seconds will be used.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

PREPARECALL

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The PREPARECALL function prepares a stored procedure call statement for execution. During preparation, the database usually precompiles the SQL statement and creates an access plan for the statement.

SQL statements may contain parameters that are evaluated at execution time. Parameters are marked by a question mark (?) within the SQL statement. The parameters are numbered in order of appearance, starting with '1'. After preparing, the application can bind host variables to the parameters. When the statement is executed later, the content of the host variables is used and sent to the database.

Related function: "PREPARESTATEMENT" on page 382.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE           PIC S9(8) BINARY.
   01 CON-HANDLE           PIC S9(8) BINARY.
   01 STMT-HANDLE          PIC S9(8) BINARY.
   01 SQL                  PIC X(nnn) VALUE IS 'CALL INOUT_PARAM(?)'.
   01 SQL-LEN              PIC S9(8) BINARY VALUE IS 30.
   01 CURSOR-TYPE          PIC S9(8) BINARY VALUE IS 1003.
   01 CONCURRENCY          PIC S9(8) BINARY VALUE IS 1007.
   01 HOLDABILITY          PIC S9(8) BINARY VALUE IS 1.
   01 RETCODE              PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-PREPARECALL ENV-HANDLE CON-HANDLE
        STMT-HANDLE SQL SQL-LEN [CURSOR-TYPE CONCURRENCY [HOLDABILITY]]
        RETCODE.
```

FUNCTION (input)
A 16-byte character field containing PREPARECALL. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by the PREPARECALL call must be specified with subsequent call to other DBCLI functions that require a statement.

SQL (input)
A character field containing the SQL statement to prepare. The field should be padded with blanks to the right, up to the length specified by the SQL-LEN parameter.

SQL-LEN (input)
A fullword binary variable containing the length of the SQL statement specified in the SQL parameter.

CURSOR-TYPE (input, optional)
A fullword binary variable containing the cursor type. This parameter is optional. If specified, the CONCURRENCY parameter must also be specified. The cursor type controls if the cursor is forward-only, or if it can be used for scrolling. The following values can be used:

CURSOR-TYPE-FORWARD-ONLY (1003):
The cursor can move only forward.

CURSOR-TYPE-SCROLL-INSENSITIVE (1004):
The cursor is scrollable but generally not sensitive to changes made by others.

CURSOR-TYPE-SCROLL-SENSITIVE (1005):
The cursor is scrollable and generally sensitive to changes made by others.

CONCURRENCY (input, optional)
A fullword binary variable containing the concurrency mode. This parameter is optional. If specified, the CURSOR-TYPE parameter must also be specified. The concurrency mode controls if the cursor is read-only, or if it can be updated. The following values can be used:

CURSOR-CONCUR-READ-ONLY (1007):
The cursor is read-only.

CURSOR-CONCUR-UPDATABLE (1008):
The cursor is updatable.

HOLDABILITY (input, optional)
A fullword binary variable containing the holdability mode. This parameter is optional. If specified, the CURSOR-TYPE and CONCURRENCY parameters must also be specified. The value is a fullword binary variable containing one of the following:

HOLD-CURSORS-OVER-COMMIT (1):
Any open cursors are kept open when a COMMIT is performed.

CLOSE-CURSORS-AT-COMMIT (2):
Any open cursors are closed when a `COMMIT` is performed.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

PREPARESTATEMENT

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The `PREPARESTATEMENT` function prepares an SQL statement for execution. It allocates a statement handle that represents the statement and sets the STMT-HANDLE parameter. The application must pass the statement handle to all subsequent functions that require a statement. An application can prepare multiple statements at a time. The application must ensure it uses the required statement handle with subsequent functions.

SQL statements may contain parameters that are evaluated at execution time. Parameters are marked by a question mark (?) within the SQL statement. The parameters are numbered in order of appearance, starting with 1. After preparing, the application can bind host variables to the parameters. When the statement is executed later, the content of the host variables is used and sent to the database.

Related functions:

- “CLOSESTATEMENT” on page 308
- “PREPARECALL” on page 381

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE          PIC S9(8) BINARY.
   01 CON-HANDLE          PIC S9(8) BINARY.
   01 STMT-HANDLE         PIC S9(8) BINARY.
   01 SQL                 PIC X(nnn) VALUE IS 'SELECT * FROM TABLE WHERE A=?'.
   01 SQL-LEN             PIC S9(8) BINARY VALUE IS 30.
   01 CURSOR-TYPE         PIC S9(8) BINARY VALUE IS 1003.
   01 CONCURRENCY         PIC S9(8) BINARY VALUE IS 1007.
   01 HOLDABILITY         PIC S9(8) BINARY VALUE IS 1.
   01 RETCODE             PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-PREPARESTATEMENT ENV-HANDLE CON-HANDLE
        STMT-HANDLE SQL SQL-LEN [CURSOR-TYPE CONCURRENCY [HOLDABILITY]]
        RETCODE.
```

FUNCTION (input)
A 16-byte character field containing PREPARESTATEMENT. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

STMT-HANDLE (output)
A fullword binary variable containing the statement handle. The STMT-HANDLE value set by the PREPARESTATEMNT call must be specified with subsequent call to other DBCLI functions that require a statement.

SQL (input)
A character field containing the SQL statement to prepare. The field should be padded with blanks to the right, up to the length specified by the SQL-LEN parameter.

SQL-LEN (input)
A fullword binary variable containing the length of the SQL statement specified in the SQL parameter.

CURSOR-TYPE (input, optional)
A fullword binary variable containing the cursor type. This parameter is optional. If specified, the CONCURRENCY parameter must also be specified. The cursor type controls if the cursor is forward-only, or if it can be used for scrolling. The following values can be used:

CURSOR-TYPE-FORWARD-ONLY (1003):
The cursor can move only forward.

CURSOR-TYPE-SCROLL-INSENSITIVE (1004):
The cursor is scrollable but generally not sensitive to changes made by others.

CURSOR-TYPE-SCROLL-SENSITIVE (1005):
The cursor is scrollable and generally sensitive to changes made by others.

CONCURRENCY (input, optional)
A fullword binary variable containing the concurrency mode. This parameter is optional. If specified, the CURSOR-TYPE parameter must also be specified. The concurrency mode controls if the cursor is read-only, or if it can be updated. The following values can be used:

CURSOR-CONCUR-READ-ONLY (1007):
The cursor is read-only.

CURSOR-CONCUR-UPDATABLE (1008):
The cursor is updatable.

HOLDABILITY (input, optional)
A fullword binary variable containing the holdability mode. This parameter is optional. If specified, the CURSOR-TYPE and CONCURRENCY parameters must also be specified. The value is a fullword binary variable containing one of the following:

HOLD-CURSORS-OVER-COMMIT (1):
Any open cursors are kept open when a COMMIT is performed.

CLOSE-CURSORS-AT-COMMIT (2):
Any open cursors are closed when a COMMIT is performed.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

RELEASESAVEPOINT

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The RELEASESAVEPOINT function removes the specified save point from the current logical unit or work.

Related function: "SETSAVEPOINT" on page 388.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE           PIC S9(8) BINARY.
   01 CON-HANDLE           PIC S9(8) BINARY.
   01 SAVEPOINT            PIC S9(8) BINARY.
   01 RETCODE              PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-RELEASESAVEPOINT ENV-HANDLE CON-HANDLE
        SAVEPOINT RETCODE.
```

FUNCTION (input)
A 16-byte character field containing RELEASESAVEPOINT. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

SAVEPOINT (input)
A fullword binary variable containing the save point to release.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

ROLLBACK

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The ROLLBACK function rolls back all changes in the database performed in the last logical unit of work. If a save point is specified, the all changes since the save point was taken are rolled back. A save point can be established using the SETSAVEPOINT function.

Related function: "COMMIT" on page 308.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 CON-HANDLE         PIC S9(8) BINARY.
   01 SAVEPOINT          PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-ROLLBACK ENV-HANDLE CON-HANDLE
        [SAVEPOINT] RETCODE.
```

FUNCTION (input)
: A 16-byte character field containing ROLLBACK. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
: A fullword binary variable containing the environment handle.

CON-HANDLE (input)
: A fullword binary variable containing the connection handle.

SAVEPOINT (input, optional)
: A fullword binary variable containing the save point to roll back to.

RETCODE (output)
: A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

SETCONNATTR

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The SETCONNATTR function allows an application to set an attribute on the connection level.

Related function: "GETCONNATTR" on page 355.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE           PIC S9(8) BINARY.
   01 CON-HANDLE           PIC S9(8) BINARY.
   01 ATTR                 PIC S9(8) BINARY.
   01 VALUE                PIC S9(8) BINARY.
   01 VALUE-LEN            PIC S9(8) BINARY.
   01 RETCODE              PIC S9(8) BINARY.
PROCEDURE
  CALL 'IESDBCLI' USING FUNC-SETCONNATTR ENV-HANDLE CON-HANDLE
       ATTR VALUE VALUE-LEN RETCODE.
```

FUNCTION (input)
: A 16-byte character field containing SETCONNATTR. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

ATTR (input)
A fullword binary variable containing the attribute ID of the attribute to set or get.

VALUE (input/output)
A field that contains the new attribute value. Some attributes use textual values, therefore you must instead use a character field.

VALUE-LEN (input)
A fullword binary variable containing the length of the VALUE field. For attributes using a fullword binary value, the length should be set to 4.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

For details of the *connection attributes* that are defined (including the read-only connection attributes for retrieving database *meta information*), see "GETCONNATTR" on page 355.

SETENVATTR

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The SETENVATTR function allows an application to set an attribute on the environment level. In general, environment attributes should be set before a connection is established using the CONNECT function.

Related function: "GETENVATTR" on page 369.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 ATTR               PIC S9(8) BINARY.
   01 VALUE              PIC S9(8) BINARY.
   01 VALUE-LEN          PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-SETENVATTR ENV-HANDLE
        ATTR VALUE VALUE-LEN RETCODE.
```

FUNCTION (input)
A 16-byte character field containing SETENVATTR. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

ATTR (input)
A fullword binary variable containing the attribute ID of the attribute to set or get.

VALUE (input/output)
A field that contains the new attribute value. Some attributes use textual values, therefore you must instead use a character field.

VALUE-LEN (input)
A fullword binary variable containing the length of the VALUE field. For attributes using a fullword binary value, the length should be set to 4.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

For details of the *environment attributes* that are defined, see "GETENVATTR" on page 369.

SETPOS

(For a "roadmap" of all DBCLI functions sorted by category, see Table 8 on page 300).

The SETPOS function allows an application to position the cursor without fetching the data. It also allows refreshing, updating, deleting, or inserting a row of data of an open cursor. Dependent on the operation, the SETPOS function uses all host variables which have been bound to the cursor using the BINDCOLUMN function. It also uses the corresponding indicator variables to indicate if a column is NULL, or is to be ignored for an update.

Rows of a cursor can only be updated if the cursor is updatable and scrollable, i.e. if the statement has been created with CURSOR-CONCUR-UPDATABLE and CURSOR-TYPE-SCROLL-INSENSITIVE or CURSOR-TYPE-SCROLL-SENSITIVE.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE           PIC S9(8) BINARY.
   01 STMT-HANDLE          PIC S9(8) BINARY.
   01 OPERATION            PIC S9(8) BINARY.
   01 ROW-NUMBER           PIC S9(8) BINARY.
   01 RETCODE              PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-SETPOS ENV-HANDLE
        STMT-HANDLE OPERATION ROW-NUMBER RETCODE.
```

FUNCTION (input)
A 16-byte character field containing SETPOS. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

STMT-HANDLE (input)
A fullword binary variable containing the statement handle.

OPERATION (input)
A fullword binary variable containing an operation code. The following operations are supported:

SETPOS-POSITION (0)
Positions the cursor to the row specified in the ROW-NUMBER parameter.

SETPOS-REFRESH (1)
Positions the cursor to the row specified in the ROW-NUMBER parameter and forces the JDBC driver to refresh the row.

SETPOS-UPDATE (2)
Positions the cursor to the row specified in the ROW-NUMBER parameter and updates the columns of the row. The SETPOS-UPDATE operation uses all host variables which have been bound to the cursor using the

BINDCOLUMN function. It also uses the corresponding indicator variables to indicate if a column is NULL, or is to be ignored for an update (INDICATE-IGNOREUPDATE).

SETPOS-DELETE (3)
Positions the cursor to the row specified in the ROW-NUMBER parameter and deletes the row.

SETPOS-INSERT (4)
Inserts a new row. The ROW-NUMBER parameter is ignored for this operation. The SETPOS-INSERT operation uses all host variables which have been bound to the cursor using the BINDCOLUMN function. It also uses the corresponding indicator variables to indicate if a column is NULL, or is to be ignored for the insert (INDICATE-IGNOREUPDATE).

ROW-NUMBER (input)
A fullword binary variable containing the absolute row number. For the SETPOS-INSERT operation the ROW-NUMBER is ignored. For all other operations it is required.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

SETSAVEPOINT

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The SETSAVEPOINT function establishes a save point in the current logical unit or work. A save point can later be used to roll back changes using the ROLLBACK function.

Related function: “RELEASESAVEPOINT” on page 384.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE         PIC S9(8) BINARY.
   01 CON-HANDLE         PIC S9(8) BINARY.
   01 SAVEPOINT          PIC S9(8) BINARY.
   01 RETCODE            PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-SETSAVEPOINT ENV-HANDLE CON-HANDLE
        SAVEPOINT RETCODE.
```

FUNCTION (input)
A 16-byte character field containing SETSAVEPOINT. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

CON-HANDLE (input)
A fullword binary variable containing the connection handle.

SAVEPOINT (output)
A fullword binary variable containing a handle of the save point that can be used later with the RELEASESAVEPOINT or ROLLBACK functions.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

SETSTMTATTR

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The SETSTMTATTR function allows an application to set an attribute on the statement level.

Related function: “GETSTMTATTR” on page 376.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE            PIC S9(8) BINARY.
   01 STMT-HANDLE           PIC S9(8) BINARY.
   01 ATTR                  PIC S9(8) BINARY.
   01 VALUE                 PIC S9(8) BINARY.
   01 VALUE-LEN             PIC S9(8) BINARY.
   01 RETCODE               PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-SETSTMTATTR ENV-HANDLE STMT-HANDLE
        ATTR VALUE VALUE-LEN RETCODE.
```

FUNCTION (input)
A 16-byte character field containing SETSTMTATTR. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

STMT-HANDLE (input)
A fullword binary variable containing the statement handle.

ATTR (input)
A fullword binary variable containing the attribute ID of the attribute to set or get.

VALUE (input/output)
A field that contains the new attribute value. Some attributes use textual values, therefore you must instead use a character field.

VALUE-LEN (input)
A fullword binary variable containing the length of the VALUE field. For attributes using a fullword binary value, the length should be set to 4.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

For details of the *statement attributes* that are defined, see “GETSTMTATTR” on page 376.

TERMENV

(For a “roadmap” of all DBCLI functions sorted by category, see Table 8 on page 300).

The TERMENV function must be the very last call to the DBCLI API. It terminates the environment, including the EZA TCP/IP interface. The TERMENV call frees the environment handle. No further calls to DBCLI functions are allowed, except the INITENV call.

Related function: “INITENV” on page 379.

```
WORKING STORAGE
   COPY IESDBCOB.
   01 ENV-HANDLE    PIC S9(8) BINARY.
   01 RETCODE       PIC S9(8) BINARY.
PROCEDURE
   CALL 'IESDBCLI' USING FUNC-TERMENV ENV-HANDLE RETCODE.
```

FUNCTION (input)
A 16-byte character field containing TERMENV. The field is left justified and padded on the right with blanks.

ENV-HANDLE (input)
A fullword binary variable containing the environment handle.

RETCODE (output)
A fullword binary variable containing a code returned by the DBCLI call. The return code parameter must always be the very last parameter.

Performance Considerations When Using the DBCLI

When using the Database Call Level Interface (DBCLI), the following performance considerations apply.

Using SSL

The use of SSL (Secure Socket Layer) adds a certain amount of overhead to all function calls that require communication with the DBCLI server (DBCliServer). The overhead is due to encryption processing for all data that is being sent or received of the network connection. The use of hardware crypto support is strongly recommended (CPACF and Crypto Express® 2/3 cards).

Pre-fetching

Usually, the result rows of a query or stored procedure call is fetched in "chunks" of multiple rows. Therefore, the data of all bound columns is transmitted in one block of data over the network connection. This process is called *pre-fetching*, and saves processing and network overhead.

- The number of rows that are being pre-fetched can be controlled via the statement attribute STMTATTR-PREFETCHROWS (0). The default is to pre-fetch 128 rows.
- Higher pre-fetching numbers may improve fetch performance, but require a higher amount of memory.
- Memory consumption is also dependent on the number of columns per row and its size.

Note: If a cursor type of CURSOR-TYPE-SCROLL-SENSITIVE (1005) is used, pre-fetching is *disabled* for this statement.

- For a cursor that is scrollable and sensitive to changes made by others, pre-fetching cannot be used, since the row data may change in the database while the row has already been pre-fetched.
- Therefore, using a cursor type of CURSOR-TYPE-SCROLL-SENSITIVE (1005) causes every row to be fetched separately at the time when the FETCH function is called.
- This may have negative performance implications when a huge number of rows are fetched.

Binding Columns

To retrieve the result data of a query or a stored procedure, you must bind host variables to columns of the cursor. When (pre-) fetching rows, the data fore all bound columns is transferred over the network. The data translation for every column of a row takes place at the DBCliServer side. The size of the data of a column is dependent on the host variable bound to the column.

For best performance, do not bind columns that you are not interested in. Also, make sure that the host variables you bind to the columns are of the right size (e.g. do not bind a character string of 200 characters if the column only is 5 characters in length).

Investigating the Cause of Errors When Using the DBCLI

If you encounter errors when using the DBCLI interface, you are recommended to perform these actions:

- Check the return codes provided by DBCLI calls. The return code is passed back in the RETCODE parameter of each function call. The RETCODE parameter must be the very last parameter on every call.
- Check the error information provided by a call to the `GETLASTERROR` function. This function will provide the SQLCODE, SQLSTATE and a textual error message. The error message may be database and JDBC driver dependent.
- It might be helpful to enable the internal DBCLI trace. The trace prints diagnosis information to SYSLST, SYSLOG, or both. You can enable the trace as follows:
  - Use the `// SETPARM [SYSTEM] CLI$TRC=nnnn` statement in the job that runs your application. The CLI$TRC system parameter can have the following values:
    - OFF (default, same as not specified) – trace is off
    - SYSLST – trace is written to SYSLST (job listing)
    - SYSLOG – trace is written to SYSLOG (console)
    - BOTH or YES – trace is written to SYSLOG and SYSLST
  - Set the environment attribute ENVATTR-TRACE in your program using a call to the `SETENVATTR` function. This environment attribute accepts the following values:
    - TRACE-OFF (0) – trace is off
    - TRACE-SYSLST (1) – trace is written to SYSLST (job listing)
    - TRACE-SYSLOG (2) – trace is written to SYSLOG (console
    - TRACE-BOTH (3) – trace is written to SYSLOG and SYSLST
- It might be helpful to enable the EZA socket API trace using the EZAAPI command (see *z/VSE TCP/IP Support* publication). This trace shows trace information regarding the EZA socket interface.

Return Codes Used by the DBCLI

The DBCLI calls return one of the following return codes in the RETCODE field:

Symbol	Value	Description
EOK	0	No error occurred.
ECDLOAD	1	A load failure has occurred when loading phase IESDBCLA.PHASE.
EHANDLE	2	An invalid handle has been passed to a function.
ENOMEM	3	Not enough memory is available.
EPARAM	4	Invalid or missing parameter.
EFUNCTION	5	An invalid function has been specified.
EINITFAILED	6	Initialization of the EZA TCP/IP interface has failed. Check if the correct TCPNAME or OPTION SYSPARM, as well as ADSNAME or SYSPARM EZA$PHS is specified, and if the TCP/IP stack is up and running. **Note:** Also check "Programming Restrictions and Requirements" on page 296 (especially the note about using the DBCLI API in CICS transactions).
ECONNFAILED	7	Failed to connect to `DBCliServer`. Check the IP address or hostname and port number.
ENOTCONN	8	No connection to `DBCliServer` exists.
ESENDFAILED	9	A send operation has failed. Most likely the connection to `DBCliServer` has broken.
ERECVFAILED	10	A receive operation has failed. Most likely the connection to `DBCliServer` has broken.
EPROTO	11	A protocol violation has been detected. Please report this error to IBM Support.
EFAIL	12	A general failure has occurred. For further details, check the error message, SQLCODE, and SQLSTATE using the GETLASTERROR function.
ESQL	13	An SQL error has occurred. For further details, check the error message, SQLCODE, and SQLSTATE using the GETLASTERROR function.
EATTRID	14	An invalid attribute has been specified.
EATTRLEN	15	The attribute length is invalid.
EATTRVAL	16	The attribute value is invalid.
EINDEX	17	The parameter or column index is out of range.
ETYPE	18	The data type is invalid.
ESTATE	19	The function can not be used with the state of the connection or statement.
ENOMOREDATA	20	No more data is available to fetch.

For most return codes the DBCLI code also provides a textual error message. You can retrieve the error message as well as SQLCODE and SQLSTATE using the GETLASTERROR function.

Chapter 23. Using the DB2-Based Connector to Access Data

This chapter describes how you use the DB2-based connector to access VSAM datasets and DL/I databases.

The DB2-based connector uses these products:
- On the z/VSE host:
  - DB2 Server for VSE

    Note: The DB2 Server for VSE Client Edition is **not** sufficient for use with the DB2-based connector, since the DB2 Server for VSE Client Edition does **not** allow DB2 Stored Procedures to be run on z/VSE.
  - The DB2 Stored Procedures facility (available with Version 6 or later of the DB2 Server for VSE).
  - The VSAMSQL CLI
  - DL/I VSE
- On the physical/logical middle-tier:
  - DB2 Connect
  - ODBC, JDBC, or CLI

For a general introduction to the DB2-based connector, see “Overview of the DB2-Based Connector” on page 83.

This chapter contains these main topics:
- “How You Use DB2 Stored Procedures” (which includes a description of Stored Procedure Servers)
- “Using DB2 Stored Procedures to Access VSAM Data” on page 395
- “Using DB2 Stored Procedures to Access DL/I Data” on page 402

How You Use DB2 Stored Procedures

A DB2 Stored Procedure is a *program* that you write yourself, and then compile and catalog into a library, and then define to DB2. Your DB2 Stored Procedures can then be called from:
- A Web Client
- The physical/logical middle-tier of a 3-tier z/VSE environment
- A local batch program.

You can write a DB2 Stored Procedure in any LE (Language Environment)-compliant language. The API (Application Programming Interface) within a DB2 Stored Procedure differs depending upon whether VSAM or DL/I data is being accessed on the z/VSE host.

These are the main advantages of using DB2 Stored Procedures:
- When the database manager is running in multiple user mode, local applications or remote DRDA applications can invoke a DB2 Stored Procedure. Since the SQL statements issued by a DB2 Stored Procedure are local to the z/VSE host, they

© Copyright IBM Corp. 2000, 2013

do not incur the high network costs of distributed statements. Instead, single network *send* and *receive* operations can be used for processing all the statements contained in a DB2 Stored Procedure.

- You can use a DB2 Stored Procedure to hide the details of the database design from application programs running on the Web Client or physical/logical middle-tier.
- If a database is modified, only the DB2 Stored Procedure (and not the application programs) needs to be modified.
- You can use a DB2 Stored Procedure to hide sensitive data from specific application programs.
- You can encapsulate business logic at the z/VSE host, instead of having to include this business logic in numerous application programs.
- It is easier to maintain an environment in which DB2 Stored Procedure applications are maintained at the z/VSE host, instead of being distributed across a number of Web Clients or physical/logical middle-tiers.

Grouping Stored Procedure Servers

The *Stored Procedure Server* is contained in a separate static or dynamic partition, and must be dedicated to a single DB2 Server for VSE. The DB2 Stored Procedures described in the previous topic run under the control of a Stored Procedure Server, which is LE-compliant.

By grouping your Stored Procedure Servers, you can distribute the database workload over multiple partitions. This might be useful if certain DB2 Stored Procedures must always have a DB2 Server for VSE available. In this case, a Stored Procedure Server group could be dedicated to the DB2 Stored Procedure. Other DB2 Stored Procedures could then share other Stored Procedure Server groups:

- Some DB2 Stored Procedures might have special requirements (for example, they require unusually large amounts of virtual storage).
- Other DB2 Stored Procedures might have to access resources that are not required by the other DB2 Stored Procedures.

The ability to group Stored Procedure Servers provides the database administrator with flexibility when defining the environment, and is also useful for system-tuning purposes.

Programming Requirements When Using DB2 Stored Procedures

On the Web Client or physical/logical middle-tier: you require JDBC or ODBC/CLI (which do not require a SQL precompiler). If however you use another programming language that contains embedded SQL to call a DB2 Stored Procedure, you *do require* an SQL precompiler.

On the z/VSE host: providing the DB2 Stored Procedure does not contain SQL-specific calls, you do not require an SQL precompiler.

The VSAMSQL CLI consists of a small object module that you must link to each of your DB2 Stored Procedures that will use the VSAMSQL CLI. This object module is a "stub" that allows your DB2 Stored Procedures to use the CLI (Call Level Interface) to access VSE/VSAM data *as if the data were relational*.

A DB2 Stored Procedure must be LE-compliant. Language Environment is the prerequisite run-time environment for applications generated using compilers that run with z/VSE.

These are the most important interfaces that you can use from your application programs:
- ODBC (Open DataBase Connectivity) on your physical/logical middle-tier platform
- CLI (Call Level Interface) in your DB2 Stored Procedure running on the z/VSE host

Using DB2 Stored Procedures to Access VSAM Data

This topic describes how you can use DB2 Stored Procedures to access VSE/VSAM data.

Although Figure 151 on page 396 shows the access to VSAM data only, you can access DB2 and DL/I data using the same DB2 Stored Procedure.

To access the mapped VSAM data, your application programs use a VSAMSQL Call Level Interface (CLI), which is based upon the IBM DB2 Call Level interface. Using the VSAMSQL CLI, your application programs can issue SQL-like calls to VSAM data from within a DB2 Stored Procedure, as described in the following topics:
- "Overview: Accessing VSAM Data via DB2 Stored Procedures" provides an overview of how DB2 Stored Procedures are used to display mapped VSAM data via the VSAMSQL CLI.
- "Using the Call Level Interface: Activities on the Requestor" on page 397 describes the activities you must perform *on the client* in order to use a DB2 Stored Procedure to display mapped VSAM data via the VSAMSQL CLI.
- "Using Call Level Interface: Activities on the z/VSE host" on page 398 describes the activities you must perform *on the z/VSE host* in order to use a DB2 Stored Procedure to display mapped VSAM data via the VSAMSQL CLI.
- "Program Flow When Using the VSAMSQL Call Level Interface" on page 399 describes the typical program flow when a DB2 Stored Procedure performs a VSAM-CLI update on mapped VSAM data.
- "SQL Statements Supported by VSAMSQL Call Level Interface" on page 400 lists the SQL statements that you can use when a DB2 Stored Procedure performs a VSAM-CLI update on mapped VSAM data.

Note: For a practical example of how an application program uses the VSAMSQL CLI, see "Step 2. Initialize the VSAMSQL CLI Environment" on page 211.

Overview: Accessing VSAM Data via DB2 Stored Procedures

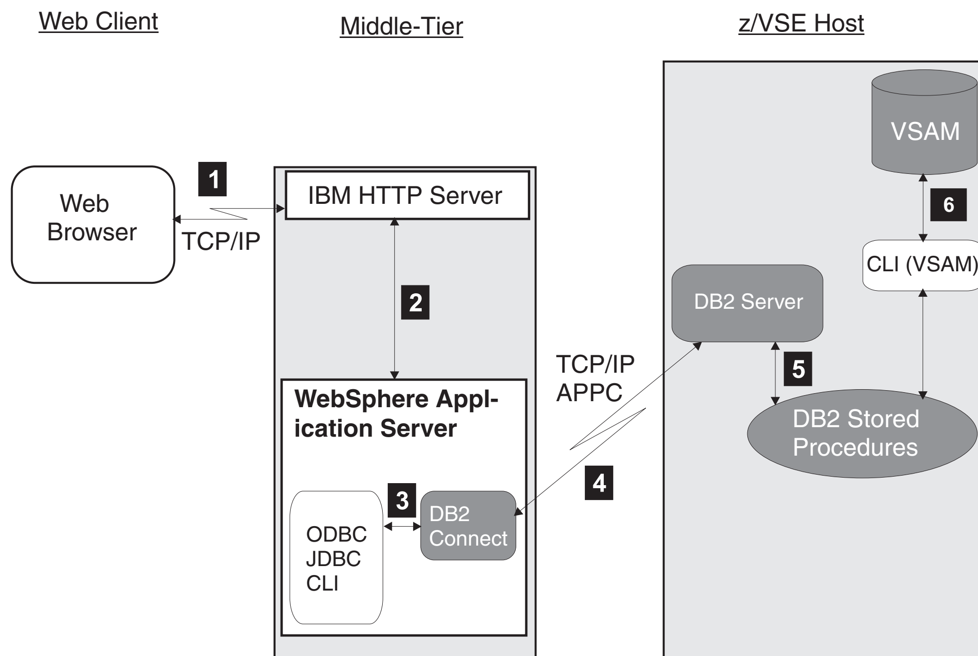


Figure 151. How You Use DB2 Stored Procedures To Access VSAM Data

The number of each list item below describes a step shown in Figure 151:

1 The clients's Web browser requests an HTML page from the IBM HTTP Server running on the physical/logical middle-tier.

2 The IBM HTTP Server calls the WebSphere Application Server for requests contained in the HTML page (for example, requests for an applet or servlet).

3 The interfaces (ODBC, JDBC, or CLI) communicate with the DB2 Server for VSE, via DB2 Connect.

4 DB2 Connect communicates with the DB2 Server for VSE, via DRDA (Distributed Relational Database Architecture). The underlying protocol used here can be either APPC or TCP/IP.

5 DB2 Server for VSE manages the execution of a DB2 Stored Procedure, using the Stored Procedure Server.

6 The DB2 Stored Procedure can now access the VSAM data stored on the z/VSE host, via VSAMSQL CLI. The DB2 Stored Procedure contains calls to the VSAM *CLI functions* (described in Table 9 on page 398), which are used to perform the access.

The process is now the reverse of steps 1 to 6. The DB2 Server for VSE passes the results back to the requester.

As shown in Figure 151, to use a DB2 Stored Procedure to access VSAM data stored on the z/VSE host, you require:

- *On the physical/logical middle-tier:*
  - A Web Server (such as the IBM HTTP Server)
  - The WebSphere Application Server
  - DB2 Connect
  - ODBC or CLI

- *On the z/VSE host:*
  – The DB2 Server for VSE
  – The VSAMSQL CLI

These are the general steps you should follow to develop application programs that access mapped VSAM data via DB2 Stored Procedures:

1. Establish DB2 connection between the requestor and z/VSE host.
2. Design your application program and the VSAM cluster to be used for storing your maps.
3. Define the maps and views for the VSAM cluster (see Chapter 12, “Mapping VSE/VSAM Data to a Relational Structure,” on page 103).
4. Write your DB2 Stored Procedure that includes the logic and requests to VSAM and DB2 (refer to the IBM publications *DB2 Server for VSE & VM, Database Administration*, SC09-2888 and *DB2 Server for VSE & VM, Application Programming*, SC09-2889).
5. Create an entry in DB2 for your DB2 Stored Procedure.
6. Write the requestor calls to the DB2 Stored Procedure.
7. Test and run your application program.

Using the Call Level Interface: Activities on the Requestor

On the requestor, you can invoke DB2 Stored Procedures via Stored-Procedure calls that are implemented in relational database interfaces such as JDBC or ODBC/CLI. Here is a summary of the stored-procedure interfaces for these relational databases:

JDBC

```
String sql="Call <proc_name> (?,?,?,?,?,?)";
statement = con.prepareCall (sql);
statement.execute ();
```

ODBC/CLI

```
CALL procedure_name (?,?,?,?,...)
SQLExecDirect() or
SQLPrepare() followed by SQLExecute()
```

Embedded SQL

```
CALL proc_name [(parm1[:parmind1],...,parmn[:parmindn])] or
CALL proc_name USING DESCRIPTOR :sqlda
```

For a detailed description of the above interfaces, refer to these documents that describe how stored procedures are called:

- Microsoft ODBC SDK Programmer's Reference
- IBM DB2 Universal Database™ Call Level Interface Guide and Reference (S10J-8159)
- IBM Embedded SQL Programming Guide (S10J-8158)
- The JDBC Data Access API, at this Web site:
  `http://www.oracle.com/technetwork/java/javase/jdbc/`

Using Call Level Interface: Activities on the z/VSE host

On the z/VSE host, a VSAMSQL Call Level Interface (CLI) provides you with C-program functions for accessing mapped VSAM data using a DB2 Stored Procedure. This interface accesses a VSAM file in the same way as a table is used in a relational database. Therefore, you must define a *relational* view for each VSAM cluster whose mapped data you wish to access using a DB2 Stored Procedure via the VSAMSQL interface.

All VSAMSQL CLI C-program functions have a prefix VSAMSQL, and have the same syntax and functionality as DB2 CLI functions (where DB2 CLI functions have a prefix SQL).

Note: To use the CLI functions, you have to include the C header file *IESVSQL.h* in your program source. The object file *IESVSQLO.Obj* must be linked to your application. Both *IESVSQL.h* and *IESVSQLO.Obj* are located in VSE library PRD1.BASE.

These CLI functions are supported:

Table 9. CLI Functions You Can Use for Accessing Mapped VSAM Data

CLI Function	Description
VSAMSQLAllocConnect *	Allocate Connection Handle
VSAMSQLAllocEnv *	Allocate Environment Handle
VSAMSQLAllocHandle	Allocates Environment, Connection, Statement, or Descriptor Handles
VSAMSQLAllocStmt *	Allocate a Statement Handle
VSAMSQLBindCol	Bind a Column to an Application Variable
VSAMSQLBindParameter	Bind A Parameter Marker to a Buffer
VSAMSQLCloseTable	Close a specified table (Cluster)
VSAMSQLColAttribute *	Return a Column Attribute
VSAMSQLColAttributes	Get Column Attributes
VSAMSQLColumns	Get Column Information for a Table
VSAMSQLDescribeCol	Return a Set of Attributes for a Column
VSAMSQLError	Retrieve Error Information
VSAMSQLExecDirect	Execute a Statement Directly
VSAMSQLExecute	Execute a Statement
VSAMSQLFetch	Fetch Next Row
VSAMSQLFreeConnect *	Free Connection Handle
VSAMSQLFreeEnv *	Free Environment Handle
VSAMSQLFreeHandle	Free Handle Resources
VSAMSQLFreeStmt *	Free (or Reset) a Statement Handle
VSAMSQLGetDiagRec	Get multiple fields settings of Diagnostic Record
VSAMSQLNumParams	Get Number of Parameters in A SQL Statement
VSAMSQLNumResultCols	Get Number of Result Columns
VSAMSQLPrepare	Prepare a Statement
VSAMSQLPrimaryKeys	Get Primary Key Columns of A Table
VSAMSQLRowCount	Get Row Count

Table 9. CLI Functions You Can Use for Accessing Mapped VSAM Data (continued)

CLI Function	Description
VSAMSQLSetParam *	Bind A Parameter Marker to a Buffer
VSAMSQLTables	Get Table Information

Note: The functions marked with an asterisk (*) have been included in more recent functions. For details, refer to the IBM publication *DB2 Universal Database Call Level Interface, Guide and Reference*, S10J-1859.

Example of the Syntax of a CLI Function – VSAMSQLCloseTable

Here is an example of how the CLI function *VSAMSQLCloseTable* is described in the online documentation.

VSAMSQLCloseTable - Close a specified table (Cluster)

Purpose VSAMSQLCloseTable() closes the specified VSAM clusters.

Syntax

```
VSAMSQLRETURN   VSAMSQLCloseTable (VSAMSQLHENV       henv,
                                   VSAMSQLCHAR*      szTableName,
                                   VSAMSQLSMALLINT   cbTableName);
```

Function Arguments VSAMSQLCloseTable Arguments

Data Type	Argument	Use	Description
VSAMSQLHENV	henv	input	Environment handle
VSAMSQLCHAR*	szTableName	input	Table name to close
VSAMSQLSMALLINT	cbTableName	input	Length of the table name or VSAMSQL_NTS

Return Codes

```
VSAMSQL_SUCCESS
VSAMSQL_ERROR
VSAMSQL_INVALID_HANDLE
```

Diagnostics VSAMSQLCloseTable SQLSTATEs

SQLSTATE	Description	Explanation
HY009	Invalid argument value	The argument was invalid

Program Flow When Using the VSAMSQL Call Level Interface

This is the typical flow of program statements, when the VSAMSQL CLI is used for mapped VSAM data:

```
VSAMSQLRETURN  rc;         // Return Code
VSAMSQLHENV    hEnv;       // Environment Handle
VSAMSQLHDBC    hDBC;       // Connection Handle
VSAMSQLHSTMT   hStmt;      // statement Handle

//allocate Environment
rc = VSAMSQLAllocHandle(VSAMSQL_HANDLE_ENV,VSAMSQL_NULL_HANDLE,&hEnv);
//allocate Connection
rc = VSAMSQLAllocHandle(VSAMSQL_HANDLE_DBC,hEnv,&hDBC);
//allocate Statement
rc = VSAMSQLAllocHandle(VSAMSQL_HANDLE_STMT,hDBC,&hStmt);

// Prepare a Statement
rc = VSAMSQLPrepare(hStmt,
     "UPDATE VSESP.USER.CATALOG/VSAM.DISPLAY.DEMO.CLUSTER/MAP1 SET EMAIL=?, AGE=?"
     "WHERE NAME=?",VSAMSQL_NTS);

// Query the number of Parameters
rc = VSAMSQLNumParams(hStmt,&Num);

// Bind local Variables/Values to the Statement
rc = VSAMSQLBindParameter(hStmt,1,VSAMSQL_PARAM_INPUT,
     VSAMSQL_C_CHAR,VSAMSQL_VARCHAR,5,0,"Hugo",5,NULL);
 ...

// Execute the Statement
rc = VSAMSQLExecute(hStmt);
 ...
rc = VSAMSQLFreeHandle(VSAMSQL_HANDLE_STMT,hStmt);
rc = VSAMSQLFreeHandle(VSAMSQL_HANDLE_DBC,hDBC);
rc = VSAMSQLFreeHandle(VSAMSQL_HANDLE_ENV,hEnv);
```

Figure 152. Typical Program Flow When Performing a VSAMSQL CLI Update

SQL Statements Supported by VSAMSQL Call Level Interface

These SQL statements are supported when you use the VSAMSQL CLI on mapped VSAM data:

```
INSERT INTO table
       (col1, col2, ...)
       VALUES (val1, val2, ...),
              (val3, val4, ...),
              ...

UPDATE table
       SET col1=val2,
           col2=val2,
           ...
       WHERE col3=val3 AND
             col4=val4 AND
             ...

DELETE FROM table
       WHERE col3=val3 AND
             col4=val4 AND ...

SELECT col1, col2, ...
       FROM table
       WHERE col3=val3 AND
             col4=val4 AND
             ...
```

The **WHERE** statement supports these compare operations:

```
= (equal to)
< (less than)
> (greater than)
<= (less or equal)
>= (greater or equal)
<> (not equal)
```

Notes:
1. You can combine multiple filters using **AND**.
2. **OR** is not supported.

You specify the table using:
- VSAM Catalog
- Cluster File-ID
- map name
- view name (optional)

Here is the syntax of this statement:

```
 MY.USER.CATALOG/MY.VSAM.CLUSTER/MYMAP1
or
 MY.USER.CATALOG/MY.VSAM.CLUSTER/MYMAP1/MYVIEW1
```

The SQL statement can contain placeholders ('**?**') for parameters. The parameters must be bound before execution of the statement. The parameters are numbered from the beginning of the statement, starting at one.

You can use placeholders for these parts:
- table name
- column names
- values

To get the result set of a select statement, you can bind local variables to the result set columns. The columns of the result set are either:
- Those which have been specified in the select statement:
  ```
  ("SELECT col1,col2,... FROM...")
  ```
- All columns of the map/view:
  ```
  ("SELECT * FROM ...")
  ```

For RRDS Clusters, a special column named **RELRECNO** is used to specify the relative record number of the record (=key), which is not part of the record itself. A **SELECT * FROM** statement automatically adds the **RELRECNO** column as the last column to the result set. You can specify the **RELRECNO** column as a filter in all supported SQL statements **("UPDATE table SET col1=val1 WHERE RELRECNO=5 ...")**.

Using DB2 Stored Procedures to Access DL/I Data

This topic describes how you can use DB2 Stored Procedures to access DL/I data. Although Figure 153 shows the access to DL/I data only, you can access VSE/VSAM and DB2 data using the same DB2 Stored Procedure.

To access DL/I data from within a DB2 Stored Procedure, your program can use the AIBTDLI interface, which accesses DL/I data *directly* using DL/I methods. You do *not* map DL/I data (as you do for VSE/VSAM data).

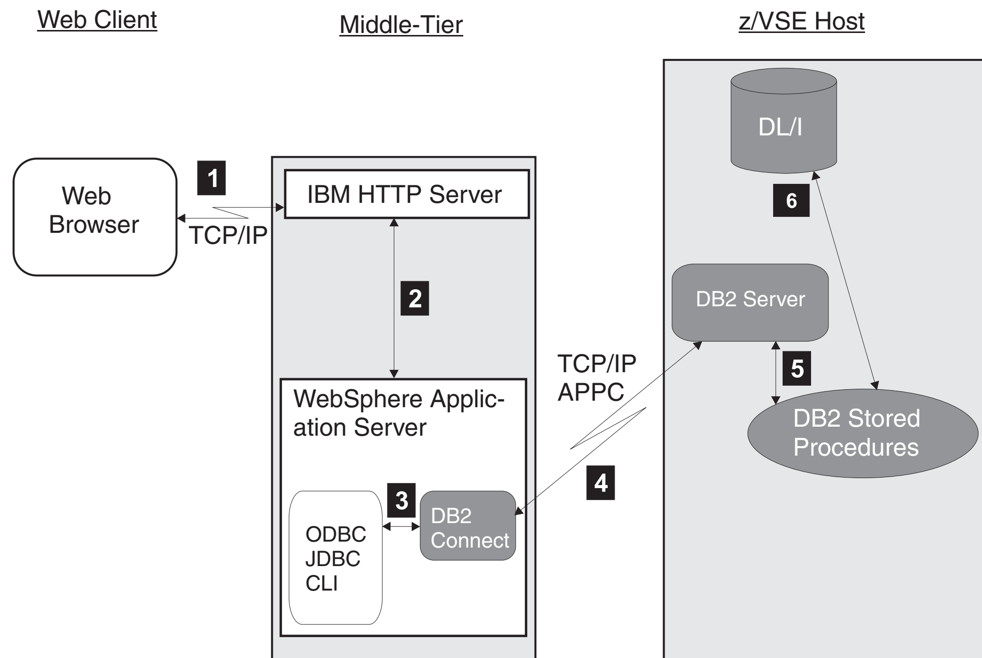


Figure 153. How You Use DB2 Stored Procedures To Access DL/I Data

The number of each list item below describes a step shown in Figure 153:

1 The clients's Web browser requests an HTML page from the IBM HTTP Server running on the physical/logical middle-tier.

2 The IBM HTTP Server calls the WebSphere Application Server for requests contained in the HTML page (for example, requests for an applet or servlet).

3 The interfaces (ODBC, JDBC, or CLI) communicate with the DB2 Server for VSE, via DB2 Connect.

4 DB2 Connect communicates with the DB2 Server for VSE, via DRDA (Distributed Relational Database Architecture). The underlying protocol used here, can be either APPC or TCP/IP.

5 DB2 Server for VSE manages the execution of a DB2 Stored Procedure, using the Stored Procedure Server.

6 The DB2 Stored Procedure can now access the DL/I data stored on the z/VSE host, via the interface AIBTDLI (see "Overview of the AIBTDLI Interface" on page 403 for details).

The process is now the reverse of steps 1 to 6. The DB2 Server for VSE passes the results back to the requester.

As shown in Figure 153 on page 402, to use a DB2 Stored Procedure to access DL/I data stored on the z/VSE host, you require:

- *On the physical/logical middle-tier:*
  - A Web Server (such as the IBM HTTP Server)
  - The WebSphere Application Server
  - DB2 Connect
  - ODBC, JDBC, or CLI
- *On the z/VSE host:*
  - The DB2 Server for VSE
  - DL/I VSE

These are the general steps you should follow to develop application programs that access DL/I data via DB2 Stored Procedures:

1. Establish DB2 connection between the requestor and z/VSE host.
2. Design your application program.
3. Write your DB2 Stored Procedure that includes the logic and requests to DL/I and VSAM or DB2 (refer to the IBM publications *DB2 Server for VSE, Database Administration*, SC092-3890 and *DB2 Server for VSE, Application Programming*, SC092-3930).
4. Create an entry in DB2 for your DB2 Stored Procedure.
5. Write the requestor calls to the DB2 Stored Procedure.
6. Test and run your application program.

Overview of the AIBTDLI Interface

The *AIBTDLI* Interface allows VSE batch programs, such as DB2 Stored Procedures, to issue DL/I calls without a DL/I batch environment having been established using DLZRRC00 or DLZMPI00.

Note: For a list of the installation requirements for using the AIBTDLI interface, see “Step 8: Customize the DB2-Based Connector for DL/I Data Access” on page 95.

AIBTDLI passes the DL/I calls to *DLZMPX00*, which connects to a running CICS/DLI online system in the form of an MPS batch task. DLZMPX00 associates each VSE batch task that uses AIBTDLI, with a DLZBPC00 mirror task on the CICS/DLI online-side, which runs on its behalf in the known MPS scheme.

The databases reside (are “opened“) at the CICS/DLI online side. All DL/I calls passed to, and results and feedback information returned from, the CICS/DLI online system are routed through DLZMPX00 via a unique XPCC connection between the VSE batch and the CICS/DLI BPC mirror task. Database access contention, resource logging, and recovery are performed on the CICS/DLI online system using existing CICS/DLI functions.

Figure 154 on page 404 illustrates the partition layout and processing flow of the three existing DL/I environments (batch, MPS batch and CICS/DLI online), and

compares them to an environment where AIBTDLI is used:

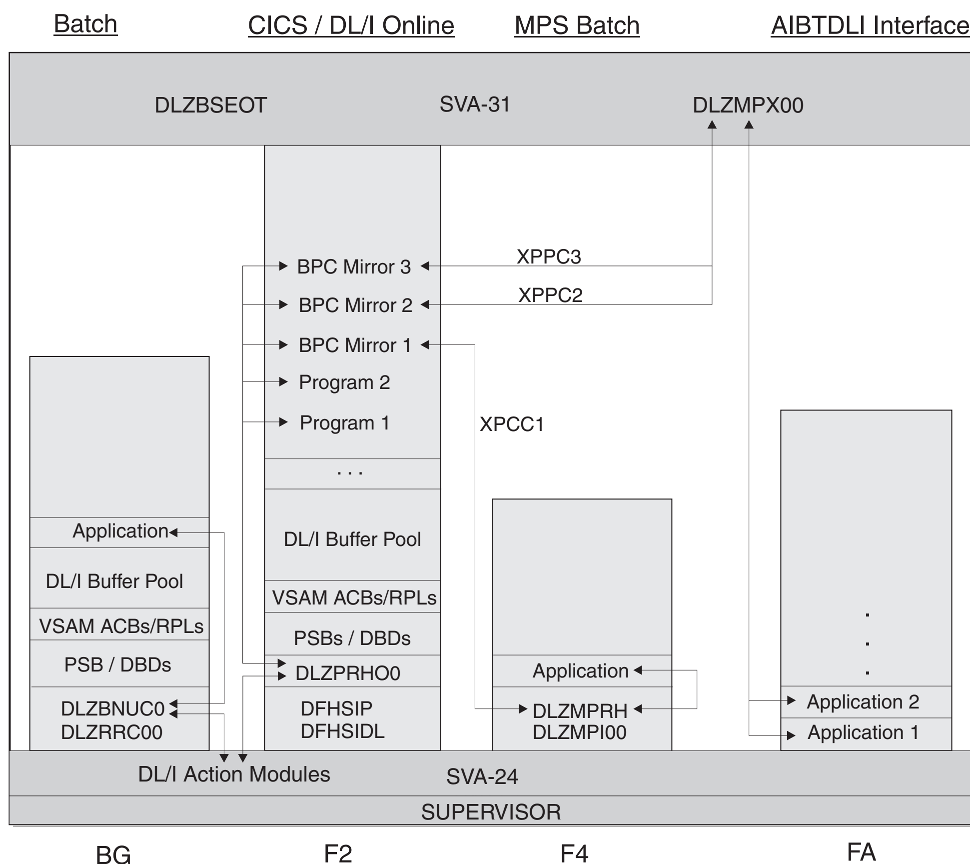


Figure 154. DL/I Partition Layout for Batch, MPS Batch, CICS/DLI Online, and AIBTDLI Interface

BG A DL/I batch environment is initialized by calling the DL/I batch root phase *DLZRRC00*. The application is loaded by DLZRRC00 and communicates with DL/I via the DL/I batch program request handler *DLZBNUC0*. All DL/I resources reside in the DL/I batch partition.

F2 A CICS/DLI online environment is established by calling the CICS root phase *DFHSIP*, which calls DL/I module *DFHSIDL* for the DL/I initialization. The online programs communicate with DL/I via the DL/I online program request handler *DLZPRHO0*, which resides in the DL/I online nucleus *DLZNUCxx*. All DL/I resources reside in the CICS/DLI partition.

F4 An MPS batch environment is initialized by calling the DL/I MPS batch root phase *DLZMPI00*. The application is loaded by DLZMPI00 and communicates with DL/I via the MPS batch program request handler *DLZMPRH*. All DL/I resources reside in the CICS/DLI online partition.

Using an XPCC connection, DLZMPRH passes DL/I calls to, and receives data from, a CICS/DLI BPC mirror task which runs on its behalf in the CICS/DLI online system.

FA In an environment where the *AIBTDLI* interface is used, a DL/I batch application can be started as the main program, which communicates with DL/I via *DLZMPX00*. All DL/I resources reside in the CICS/DLI online partition.

Using an XPCC connection, DLZMPX00 passes DL/I calls to, and receives data from, a CICS/DLI BPC mirror task which runs on its behalf in the CICS/DLI online system. The processing flow is similar to the MPS batch environment. The difference is that the initialization of a DL/I batch system is *not* required, and multiple batch programs (tasks) can use DLZMPX00 at the same time.

Creating Programs That Use AIBTDLI

The AIBTDLI interface is used by VSE batch programs. Your programs can be started as the main task, or attached as a subtask. Up to 128 (sub)tasks can use AIBTDLI at the same time.

Programs using AIBTDLI can be written in COBOL for VSE, PL/I for VSE, or Assembler. They can take any amode/rmode characteristics. All types of existing DL/I call functions are supported:

- PCB
- GET-type
- ISRT
- REPL
- DLET
- CHKP
- TERM

In addition, two new calls are provided which allow database changes to be backed out, providing they have not been committed by a DL/I CHKP or TERM call, or implicitly by task termination:

ROLB The Roll Back call is used to dynamically back out database changes up to the last syncpoint, and return control to the program.

ROLL The Roll call is used to dynamically back out database changes up to the last syncpoint, and abnormally terminate (CANCEL) the program.

The ROLB and ROLL calls are coded like a DL/I TERM call. No further parameters are required.

For a description of the general call format of the AIBTDLI interface, refer to “Invoking the AIBTDLI Interface” on page 406.

From a DL/I view, applications using AIBTDLI are implemented in the same way as CICS/DLI online programs that use the CALL interface. To access a DL/I database, your applications:

- Must start with a DL/I scheduling call, which builds up a connection between the user program (task) and the CICS/DLI MPS system and schedules the PSB. You can serially schedule more than one PSB. Before scheduling a new PSB, you must release the previous PSB using a DL/I termination call.
- Can issue any type of DL/I database calls.
- Should end with a DL/I termination call, which ends the connection between the user program (task) and the CICS/DLI MPS system, releases the PSB, frees all DL/I resources owned by this task, and provides for commit and syncpoint processing.

In order to check DL/I response information as returned in the AIB (described in “Return and Status Codes” on page 408) you must include the copybook DLIAIB.

You should be aware of the following differences to normal DL/I batch applications:

COBOL and PL/I programs

- The ENTRY (COBOL) or PROCEDURE (PL/I) statement no longer requires a reference to the PCB addresses.
- The pointers to the PCBs are instead obtained through a scheduling call in the same way as in a CICS/DLI online program.

PL/I programs

- The AIBTDLI interface has to be declared as an assembler language entry point:
```
DCL AIBTDLI ENTRY OPTIONS(ASM);
```
- PSBs do not need to be generated with LANG=PLI.

The sample programs DLZHLA80, DLZAIC50 and DLZAIP50 show how you can write a program that uses the AIBTDLI interface, when the Assembler, COBOL or PL/I language is used, respectively.

Invoking the AIBTDLI Interface

You can use the AIBTDLI interface in COBOL, PL/I, and assembler programs. The format and parameters of this interface are equivalent to the respective CBLTDLI, PLITDLI, or ASMTDLI interface, as described in *DL/I CALL and RQDLI Interfaces*. The following description shows the general format of the types of DL/I calls, and explains the scheduling call's new parameters.

Note: For assembler programs:

- You must provide an 18-fullword register save area in register 13.
- Register 1 must point to the parameter list.

Format of the Scheduling Call

The general format of the scheduling call is for:

COBOL:
CALL 'AIBTDLI' USING [parm-count,] 'PCB ',psbname, **aibparm[,destination]**.

PL/I: CALL AIBTDLI (parm-count,'PCB ',psbname, **aibparm[,destination])**;

ASSEMBLER:
CALL AIBTDLI

Here is a description of the new parameters. Existing parameters retain their previous functionality.

aibparm
is the name of a fullword to which DL/I returns the address of the *Application Interface Block (AIB)*. Use of this parameter is mandatory. The AIB is a new control block used to pass to the user the address of the PCB list and the maximum length of the I/O area following a scheduling call, the return code after each DL/I call, and pointers to a message area and a partition list in the event of an error. Details on the format and usage of the AIB are shown in "Format of the AIB – User Section" on page 408. The AIB is equivalent to the existing online UIB control block, as documented in *DL/I CALL and RQDLI Interfaces*.

After a successful scheduling call, the field *AIBPCBAL* contains the address of the PCB list.

destination
denotes the target system, where the scheduling (and all subsequent) calls should be processed. As described above, DL/I calls entered via AIBTDLI are routed to an active CICS/DLI MPS system. When MPS has been activated in more than one CICS/DLI partition at the same time, the destination specification allows the selection of the MPS subsystem to which the DL/I calls should be directed.

PARTID=xx
... xx denotes the MPS partition, where the DL/I calls should be processed

APPLID=yyyyyyyy
... yyyyyyyy denotes the CICS (generic) applid, where the DL/I calls should be processed

A destination specification is not required, if only one MPS system is running. For more information on how to select the correct target system, refer to “Scheduling with Single and Multiple MPS Systems” on page 409.

Format of the Database Call

The general format of the database call is for:

COBOL:
CALL 'AIBTDLI' USING [parm-count,]call-function, db-pcb-name,i/o-area[,ssa...].

PL/I: CALL AIBTDLI (parm-count,call-function,db-pcb-name, i/o-area[,ssa...]);

ASSEMBLER:
CALL AIBTDLI

Existing parameters retain their previous functionality.

Format of the Termination Call

The general format of the termination call is for:

COBOL:
CALL 'AIBTDLI' USING [parm-count,]'TERM'.

PL/I: CALL AIBTDLI (parm-count,'TERM');

ASSEMBLER:
CALL AIBTDLI

The function code 'TERM' can be abbreviated to 'T '.

Format of the Roll Back Call

The general format of the Roll Back call is for:

COBOL:
CALL 'AIBTDLI' USING [parm-count,]'ROLx'.

PL/I: CALL AIBTDLI (parm-count,'ROLx');

ASSEMBLER:
CALL AIBTDLI

Compiling and Link-Editing Your Programs

The compile and link requirements for DL/I programs implemented in COBOL, PL/I, and Assembler, are described in the *DL/I Release Guide*. However, for PL/I programs, there are certain differences between the information provided in the *DL/I Release Guide* and what you must specify in order to use the AIBTDLI interface:

- Your programs should *not* be compiled with * PROCESS SYSTEM(DLI);
- You should exclude these two linkage-editor statements:

```
INCLUDE IBMRPJRA
ENTRY PLICALLB
```

The sample programs DLZHLA80, DLZAIC50 and DLZAIP50 illustrate how you can compile and linkedit a DL/I program using the AIBTDLI interface, in each of the languages COBOL, PL/I, and Assembler.

Return and Status Codes

For all types of calls, DL/I returns response and error information in the new *Application Interface Block (DLIAIB)*, which you use in the same way as the previous online User Interface Block (DLIUIB). For DL/I database calls, status codes are passed back using the PCB (as was done previously).

After each DL/I call, your checking of the information returned should start with a check of the AIB. If the AIB return code does not suggest that an error has occurred, you can then examine the PCB status code.

Format of the AIB – User Section

```
DLZAIB   DSECT
AIB      DS    0F          START OF DSECT
AIBPCBAL DS    A           PCB ADDRESS LIST
AIBRCODE DS    0XL2        DL/I RETURN CODES
AIBFCTR  DS    X           RETURN CODE
AIBDLTR  DS    X           ADDITIONAL INFORMATION
         DS    2X          RESERVED
AIBMSGPT DS    A           POINTER TO ERROR MSG AREA
AIBPLPT  DS    A           POINTER TO LIST OF PARTIDS
AIBIOLM  DS    F           MAX. LENGTH OF IOAREA
         DS    2F          RESERVED
AIBLEN   EQU   *-AIB       LENGTH OF USER AIB
```

The AIB return code is passed back in the two-byte field *AIBRCODE*, which can take the same values as UIBRCODE in a CICS/DLI online program. For a list of all possible return codes of UIBRCODE (=AIBRCODE), refer to *DL/I Messages and Codes* and the *DL/I Release Guide*.

In addition to the existing return codes mentioned above, the new codes *X'080A'* and *X'FF00'* have been introduced. For details about these codes, refer to the *DL/I Release Guide*.

For code *X'FF00'*, also see "How Return Code X'FF00' Is Used" on page 409 (below).

After having inspected the feedback information in the AIB, the DL/I PCB status code should be checked. A list of all PCB status codes can be found in *DL/I Messages and Codes*. No new PCB status codes have been introduced.

The user section of the AIB will be delivered as member DLIAIB in an Assembler, COBOL and PL/I version.

How Return Code X'FF00' Is Used

When X'FF00' is returned in AIBRCODE, the AIBTDLI interface has detected an unrecoverable error. At such an event it ends the connection between the user task and the CICS/DLI MPS system, releases the PSB and frees all DL/I resources acquired by this task. Error handling for an X'FF00' situation internally triggers a DL/I termination call, which provides backout and syncpoint processing for the PSB which has been scheduled.

The reason for the X'FF00' situation is explained in a message, which DL/I writes to the console. The address field *AIBMSGPT* points to a storage area containing the message, which is formatted as follows:

- A 2-byte LL field. LL is the length of the message without the length of the LLBB field.
- A 2-byte BB field, set to binary zero.
- A variable length field containing the text of the message.

Errors That Do Not Produce a Return Code

In the following situations, DL/I is unable to return to the caller and pass back response information through the AIB:

- Load for DLZMPX00 has failed (message DLZ150I).
- Non-compatible environment (message DLZ151I).
- DL/I exit routine DLZBSEOT is not in the SVA (message DLZ152I).
- GETVIS for the AIB could not be obtained (message DLZ153I).
- DL/I subsystem registration has failed (message DLZ134I).
- AIB parameter missing or invalid (message DLZ154I).

In these cases the according error message is written to the console and the task abnormally terminated (canceled).

Scheduling with Single and Multiple MPS Systems

When only one CICS/DLI MPS system is running, the AIBTDLI interface routes a scheduling request to this MPS system. In this case, you are not required to provide a destination specification.

If MPS has been started in more than one CICS/DLI partition at the same time, you must use a destination specification as described in “Format of the Scheduling Call” on page 406. The destination specification allows the AIBTDLI interface to distinguish between the different MPS systems.

After a successful scheduling call, DL/I returns X'0000' in AIBRCODE.

When a scheduling error has occurred, AIBRCODE contains either:

- One of the existing return codes, as documented in *DL/I Messages and Codes* and the *DL/I Release Guide*.
- Return code X'FF00' and a pointer to an error message in AIBMSGPT. The message provides an explanation of why the scheduling has failed.

In the second case (return code X'FF00' and a pointer to an error message in AIBMSGPT), the explanation may be that DL/I was not able to find a suitable CICS/DLI MPS target system. You should then consider these situations:

- Message DLZ089I ('... MPC NOT ACTIVE OR ENDING') is returned in AIBMSGPT. This means that either an MPS system has not been started, or the destination specification passed as 'PARTID=' and/or 'APPLID=' parameter does not match with one of the MPS systems that are currently running.
- Message DLZ145I ('MORE THAN ONE MPS ACTIVE ...') is returned in AIBMSGPT. When MPS is active in more than one CICS/DLI partition, DL/I might not be able to determine the target system. This occurs when either a destination specification is missing in the scheduling call, or 'APPLID=' has been entered, but more than one CICS/DLI MPS system is running with the same CICS (generic) applid.

  If DLZ145I is returned, *AIBPLPT* points to a list of partition ids. The partition ids denote the partitions where an active CICS/DLI MPS system has been found, for the destination specification passed in the scheduling call:
  - If no destination has been specified, all active MPS systems are shown.
  - If 'APPLID=' has been specified, only those MPS systems are shown which are running under the searched-for CICS applid.

  Up to 10 partition ids are returned. They are represented through a list of 2-byte character fields, separated by a comma. A missing comma marks the end of the list.

  The partition ids returned via AIBPLPT can then be used to define/adapt the destination specification for a retry of the scheduling call.

Task Termination and Abend Handling

VSE task termination calls the DL/I task termination exit DLZBSEOT for tasks, which have used the AIBTDLI interface to perform final DL/I cleanup processing. This implies an internal DL/I termination call, if a PSB is still scheduled, because:

- You have not coded a regular DL/I termination call, before terminating your program.
- A termination call could not be given, when a program abend has occurred.

The internal DL/I termination call ends the connection between the task and the CICS/DLI MPS system, releases the PSB, and frees all DL/I resources acquired by this task. For a VSE normal task termination, commit and syncpoint processing is performed. For a VSE abnormal task termination, backout and syncpoint processing is performed.

The AIBTDLI interface does not handle any program errors or abend conditions. Your programs must set up and contain their own exits for dealing with, for example, AB or PC type of abends. Assembler programs may use STXIT linkage, COBOL, and PL/I programs may run with the LE TRAP runtime option.

Note: VSE considers program failures that are handled by exit routines, as “normal” processing conditions. When a program ends with a “normal” processing condition, VSE normal task termination takes place. Only unhandled program failures will lead to a VSE abnormal task termination, and backout processing.

Messages and Return Codes

The AIBTDLI interface reuses a subset of the messages available for the existing MPS batch function. Following each DL/I call, the AIBTDLI interface receives a return code from the CICS/DLI online system. These messages and return codes are described in *DL/I Messages and Codes* and the *DL/I Release Guide*.

All messages are written to the console. Error messages are also passed to the user program via the field AIBMSGPT, return codes are stored in AIBRCODE. See "Return and Status Codes" on page 408 for details.

Chapter 24. Using SOAP for Inter-Program Communication

This chapter describes how you use the Simple Object Access Protocol (abbreviated to SOAP) to send and receive information between CICS programs and other modules, over the Internet.

It contains these main topics:

- "Overview of z/VSE Support for Web Services and SOAP" on page 414
- "Overview of the SOAP Syntax" on page 414
- "Overview of Web Service (SOAP) Security" on page 415
- "How the z/VSE Host Can Act As the SOAP Server" on page 418
- "How the z/VSE Host Can Act As the SOAP Client" on page 420
- "How the IBM-Supplied SOAP Control Blocks Are Used" on page 421
- "Configuring the z/VSE SOAP Engine" on page 426
- "Mapping Long-Names to Short-Names" on page 428
- "Description of the IBM-Supplied SOAP Service (getquote.c)" on page 428
- "Description of the IBM-Supplied SOAP Client (soapclnt.c)" on page 430
- "Using a Java SOAP Client" on page 432
- "Running the IBM-Supplied SOAP Sample" on page 433
- "Writing Your Own SOAP Programs" on page 436

For detailed information about SOAP, you might go to the Apache SOAP documentation Web site, whose URL is:

```
http://ws.apache.org/axis/
```

Notes:

1. The implementation of SOAP for z/VSE is for use with the CICS Transaction Server for VSE/ESA only.
2. The implementation of SOAP for z/VSE does not require the use of either:
   - UDDI (Universal Description, Discovery, and Integration),
   - WSDL (Web Services Description Language).

Related Topic:

For details of how to ...	Refer to ...
use the *CICS2WS Toolkit* to create Web Services proxy code from CICS programs	http://www.ibm.com/systems/z/os/zvse/products/connectors.html

Note: The *CICS2WS Toolkit* is a development tool that helps you to use Web Services with your existing CICS programs. The tool reads WSDL files and Copybooks and creates proxy code that you use as a layer between your existing programs and the z/VSE SOAP Engine. The proxy code is generated as an *assembler program*. Therefore, you do not require a COBOL or PL/I compiler. It allows a z/VSE system to act as a Web Service provider (server) and as a Web Service requestor (client). It can create proxy code for both Web Service provider and Web Service requestor functions.

© Copyright IBM Corp. 2000, 2013

Overview of z/VSE Support for Web Services and SOAP

SOAP is a standard, XML-based, industry-wide protocol that allows applications to exchange information over the Internet via HTTP.

XML is a universal format that is used for structured documents and data on the Web. It is independent of both the Web client's operating-system platform and the programming language used. HTTP is supported by all Internet Web browsers and servers.

SOAP combines the benefits of both XML and HTTP into one standard application protocol. As a result, you can send and receive information to/from various platforms.

Using Web browsers, you can view information contained on Web sites. However, using SOAP you can:

- combine the contents of *different* Web sites and services.
- generate a complete view of all the relevant information.

z/VSE supports the SOAP protocol and therefore allows you to implement Web services.

An example of using SOAP might be when a travel agent requires a combined view of the Web services covering hotel reservation, flight booking, and car rental. After the travel agent has entered the required data, all three Web services from the three different providers would be processed in one transparent step. This is an example of how a "Business-to-Business" (B2B) relationship can be implemented.

For details of how to implement SOAP in your z/VSE system, see Chapter 24, "Using SOAP for Inter-Program Communication," on page 413.

Overview of the SOAP Syntax

You do not usually need to concern yourself with the tagging described here, since it is *automatically* generated by either the:

- SOAP client, and converted to native data by the SOAP server.
- SOAP server, and converted to native data by the SOAP client-processor.

However, for debugging purposes you might require a knowledge of the SOAP tagging. In this case for detailed information, you should refer to the Apache Web site whose URL is given in the previous topic. The information below provides you with an overview only.

A SOAP message is a standard XML document containing these main parts:

- A SOAP envelope, that defines the content of the message.
- A SOAP header (optional), that contains header information.
- A SOAP body, that contains call and reply information.

These are the types of element used for the SOAP message:

- The <Envelope> element is the root element of a SOAP message, and defines the XML document to be a SOAP message.
- The <Header> element can be used to include additional, application-specific information about either the SOAP message or security-specific information. The information here is user-defined. For example, it might be used to define the language used for the message.

- The <Body> element is used to define the message itself.
- The <Fault> element can be optionally used within the <Body> element, and is used to supply information about any errors that might have occurred, when the SOAP message was processed.

Here is an example of the SOAP syntax:

```
<soap:Envelope>
  <soap:Header>
   ...
  </soap:Header>
  <soap:Body>
    <GetStock>
        <Company>IBM</Company>
    </GetStock>
  </soap:Body>
</soap:Envelope>
```

The above example shows a SOAP XML document used for requesting the IBM share price. It is, of course, much simplified.

Overview of Web Service (SOAP) Security

Web Service security can be described under:
- Transport-layer security
- Message-layer security.

Both transport-layer and message-layer security provide security features for:
- Authentication/authorization
- Data encryption and signatures.

Comparison of Transport-Layer Security and Message-Layer Security

Transport-layer security secures the network communication between the communication partners by encrypting the data that is being transmitted over the network. In addition, data integrity, authentication, and confidentiality can be achieved. Transport-layer security typically uses digital signatures, PKI certificates, and secure hash functions to prevent messages from being “camouflaged”, passwords from being hacked, and transactions from being denied.

In situations where an environment consists of several hops, the communication between each hop has to be considered separately in terms of transport-layer security:

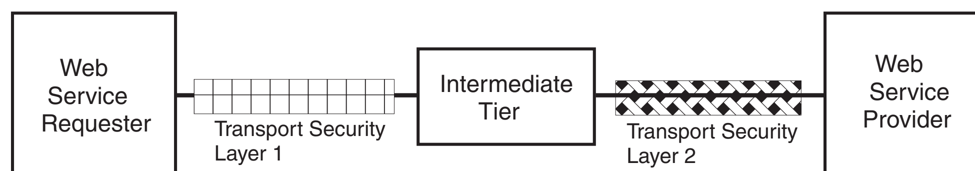


Figure 155. Connections Between Web Service Requester and Web Service Provider

As shown in Figure 155, the connections between each hop might use *different* transport-layer security methods (or even *no* transport security for some connections). Transport-layer security does not “span” multiple hops. This means, an intermediate hop might be able to read the message. To achieve end-to-end

security, you must therefore use *message-layer security*. Using message-layer security, the message itself is secure and does not change when sent over multiple hops.

Transport-layer security can be implemented using any of the industry-wide protocols, such as:
- SSL (Secure Socket Layer), which is denoted by HTTP**S**.
- VPN/IPSec (which is transparent to applications).

Message-layer security includes security-related information in the SOAP message (or more specifically, within the SOAP header).

Using Authentication With Web Service Security

Using authentication allows a service provider to check who is using the requested service. In addition, the service provider may use this information to execute the service under a specific user-ID, with its associated access rights (authorization).

To fully understand authentication and authorization, it is important to understand the following concepts:

Authentication
: The process of identifying an individual using the credentials of that individual.

Authorization
: The process of determining whether an authenticated client is allowed to access a resource or perform a task within a security domain. Authorization uses information about a client's identity and/or roles to determine the resources or tasks that a client can perform.

Credentials
: A set of claims used to prove the identity of a client. They contain an identifier for the client and a proof of the client’s identity such as a password. They may also include information, such as a signature, to indicate that the issuer certifies the claims in the credential.

Identification
: The use of an identifier that allows a system to recognize a particular subject and distinguish it from other users of the system.

There are two possible methods of performing authentication:
- *Transport-layer authentication*. Here, the transport layer carries information about who is requesting the service. The possible implementations are:
  - HTTP Authentication (Basic and Digest Access Authorization, see RFC 2617).
  - The use of SSL Client Authentication with SSL/HTTPS.
- *Message-layer authentication*. Here, the SOAP message itself carries information about who is requesting the service. The possible implementations are:
  - Direct authentication, using plain text passwords or a password digest.
  - Brokered Authentication, using a X.509 Certificate, Kerberos, Security Token Services, or SAML Assertion. Brokered Authentication using a X.509 Certificate carries the X.509 Certificate as part of the SOAP header. Here is an example:

```
<soap:Header>
  <Security xmlns="...secext-1.0.xsd>
    <BinarySecurityToken EncodigType= "wsse:Base64Binary"
                ValueType= "wsse:X509v3">
      MIICuzCCAiQCBF...
```

```
        ...
      </BinarySecurityToken>
    </Security>
    ...
```

Direct authentication defines *two* ways of transporting the password:

- *Plain text password*, in which `UsernameToken` is used to transport the actual password. If you use plain-text password configuration, you must use a secure transport method (such as HTTPS) . Here is an example:

  ```
  <soap:Header>
    <Security xmlns="...secext-1.0.xsd>
      <UsernameToken>
        <Username>John Smith</Username>
        <Password>Pass12wd</Password>
      </UsernameToken>
    </Security>
    ...
  ```

- *Password digest*, in which the communicating parties (the requester and the service) use an insecure transport channel. Steps must be taken to protect the passwords from being exposed to others. Here, the requester creates a `digest` of the actual password that is concatenated with a set of random bytes (field `nonce`) and another value that is dependent on the creation-time (field `created`). This digest is computed as follows:

  ```
  digest = Base64_encode(SHA-1(nonce+created+password))
  ```

  To authenticate the request, the service computes the digest value using the password bound to the received `username`. It compares the received `digest` value with the computed `digest` value. Here is an example:

  ```
  <soap:Header>
    <Security xmlns="...secext-1.0.xsd>
      <UsernameToken>
        <Username>John Smith</Username>
        <Password Type="...#PasswordDigest">AFHHF23wger=</Password>
        <Nonce>ksSDGFljdfD=</Nonce>
        <Created>2010-07-15T07:12:19.573Z>/Created>
      </UsernameToken>
    </Security>
    ...
  ```

From z/VSE 4.2 onwards, z/VSE supports:

- Transport-layer authentication using:
  - HTTP authentication (Basic and Digest Access Authorization).
  - SSL Client Authentication with HTTPS.
- Message-layer authentication using:
  - a `UsernameToken` (plaintext password or password digest).
  - an X.509 Certificate (`BinarySecurityToken`).

How the z/VSE Host Can Act As the SOAP Server

Figure 156 shows how SOAP can be used in a CICS environment when the z/VSE host acts as the SOAP server that provides *SOAP services* (in the z/VSE environment, CICS User Transactions).

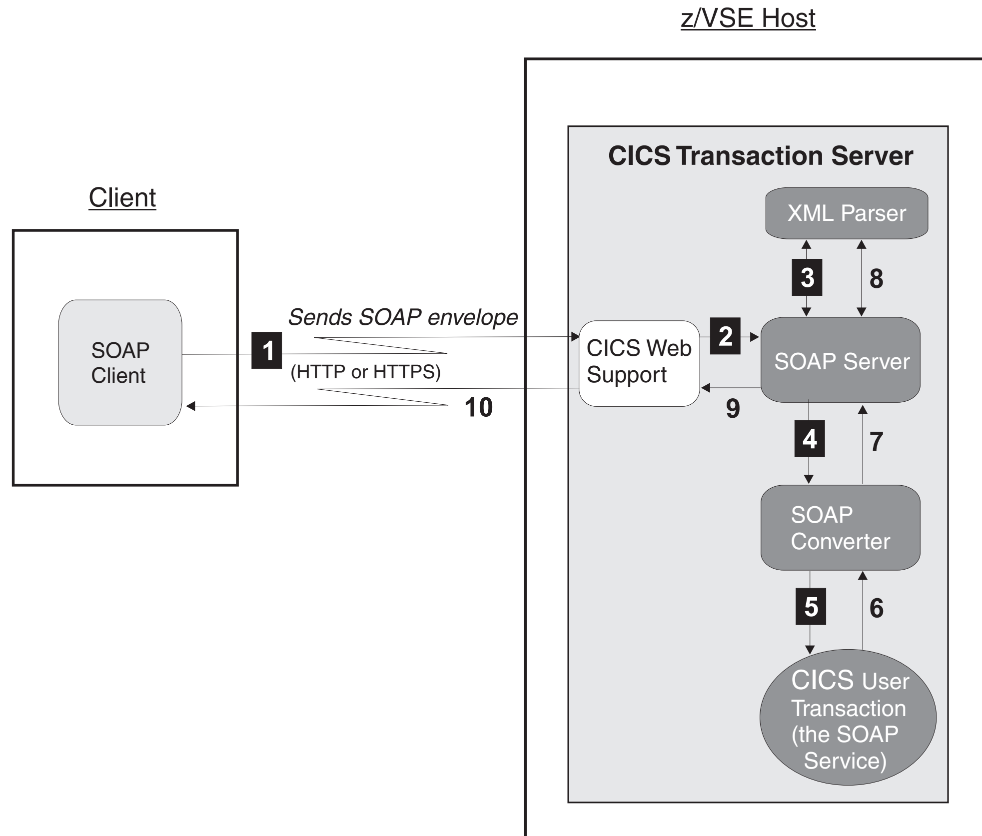


Figure 156. How SOAP Is Used When the z/VSE Host Acts As SOAP Server

1 The SOAP client (for example a platform using Microsoft .NET, IBM Websphere, Apache SOAP, or AXIS) sends a SOAP envelope (in XML format) to the SOAP server running under CICS. The SOAP envelope is sent via the CICS Web Support (CWS) component of the CICS Transaction Server.

2 CWS forwards the SOAP envelope (in XML format) to the SOAP server running under CICS.

3 The SOAP server forwards the SOAP envelope to the XML parser, also running under CICS. The XML parser then parses the SOAP envelope from textual XML format into a tree-representation of the data. For example, if the data is to be processed by a C program, the SOAP envelope would be converted to a C program structure (with pointers) so that a C program running on the z/VSE host could process the data, and returns this parsed XML tree to the SOAP server.

4 The SOAP server forwards the parsed XML tree to the SOAP converter running under CICS. The SOAP converter de-serializes (decodes) the parameter sub-tree contained in the parsed XML tree, and converts the parameter sub-tree into a binary representation. IBM-supplied SOAP decoder is named IESSOAPD.

5 The SOAP converter forwards the binary representation of the parameter sub-tree to the CICS User Transaction running on the z/VSE host (the SOAP service), via the communication area (COMMAREA) of the CICS User Transaction. The CICS User Transaction then processes the data.

The reply is then sent from the CICS User Transaction (the SOAP service) back to the SOAP client, using the reverse of the above steps (that is, steps 6 to 10). The reply is sent via the Communication area back to the SOAP converter, which serializes the parameters and returns them to the SOAP server. The SOAP server uses the XML parser to convert it from a tree-representation of the data to textual XML. The SOAP server then creates a SOAP envelope, which is then sent back (via HTTP or HTTPS) to the SOAP client. The SOAP client can then convert the SOAP envelope to its own native data format, and process the reply.

Using Web Service Security Features When z/VSE Acts As the SOAP Server

These are the areas you should consider:

- *Transport-Layer Encryption*. When transport-layer authentication is used, z/VSE acts as an HTTP server. This is implemented using CICS Web Support (CWS) as the HTTP server. CWS passes the SOAP request to the z/VSE SOAP Engine for further processing. Since CWS implements support for HTTP over SSL (HTTPS), the SOAP Engine inherits the security features from CWS. To use HTTPS, you must:
  - Configure `TCPIPSERVICE` in CICS for use with SSL.
  - Create the required keys and certificates.
- *Transport-Layer Authentication*. CWS supports SSL client authentication (HTTPS), as well as HTTP Basic Authentication, so the z/VSE SOAP Engine inherits the security features from CWS. To force a client to use HTTP basic authentication, you need to configure the TCPIPSERVICE to use the CICS provided converter program DFH$WBSB (specify `URM=DFH$WBSB`). In addition, the z/VSE SOAP Engine extracts authentication information (user-ID and password for HTTP basic authentication, or the mapped user-ID for SSL client authentication). This information can be used by the converter code to check if transport layer authentication was used. If authentication was not used, the converter code might reject the request.
- *Message-Layer Authentication*. To support message layer authentication, the z/VSE SOAP Engine (that is, the z/VSE SOAP Server) extracts the authentication token from the SOAP header after parsing the XML data stream. In case of `UsernameToken`, the user-ID and password have to be verified against a local identity store. To do this, the identity store must be able to compare the plain text password of password digest against its stored password. If user authorization is to be additionally performed, a user mapping must be performed to map the received username to a z/VSE user-ID. Also, a CICS SIGNON has to be performed using the mapped user, to allow the transaction to run under that user. In addition to `UsernameToken`, the use of certificates for authentication is possible. In this case, the converter code maps the received certificate to a z/VSE user. z/VSE supports this functionality as part of its support for SSL client authentication.

You should be aware that the z/VSE SOAP Engine does not itself perform the authentication. Instead, it simply extracts the security information and passes it to the converter code or user application. It is the user application's responsibility to perform authorization checking or signon processing.

How the z/VSE Host Can Act As the SOAP Client

Figure 157 shows how SOAP can be used in a CICS environment, when the z/VSE host acts as the SOAP client:

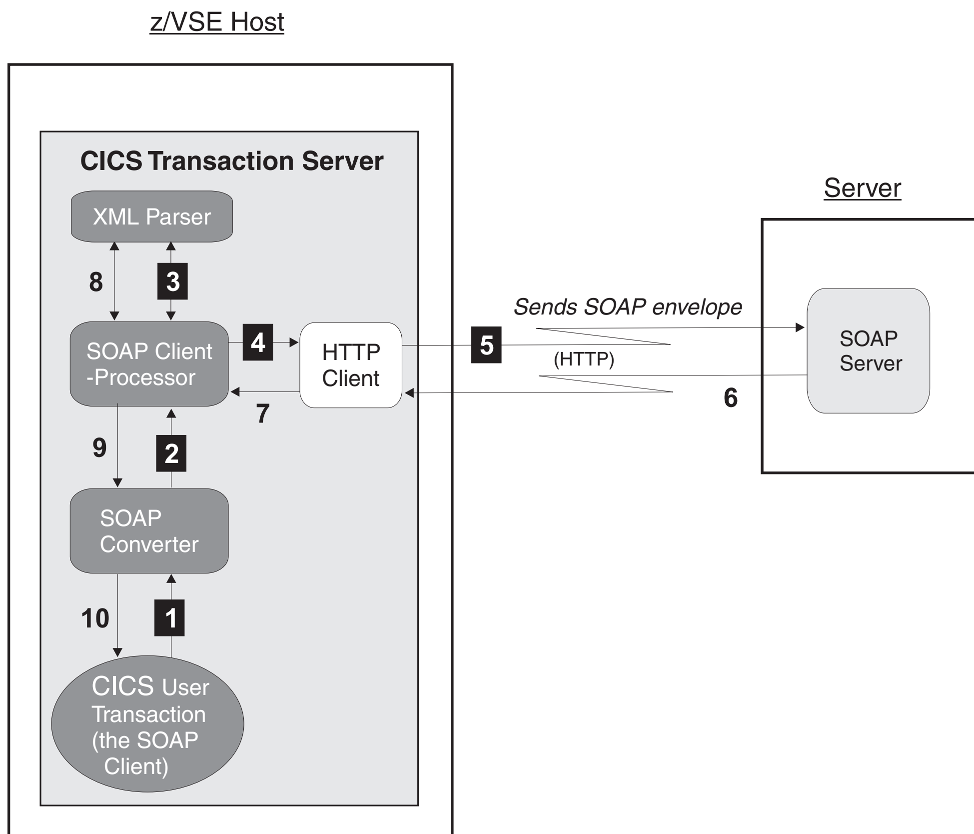


Figure 157. How SOAP Is Used When the z/VSE Host Acts As SOAP Client

1 The CICS User Transaction running on the z/VSE host (the SOAP client) sends the binary representation of the parameters to the SOAP converter. This is done via the communication area (COMMAREA) of the SOAP converter.

2 The encoder-part of the SOAP converter (IESSOAPE) serializes (encodes) the binary representation of the parameters, into an XML tree. The SOAP converter forwards the XML tree to the SOAP client-processor running under CICS.

3 The SOAP client-processor generates the SOAP envelope, and forwards it to the XML parser running under CICS. The XML parser then converts the XML tree into a textual XML format of the SOAP envelope. The XML parser returns the textual XML format to the SOAP client-processor.

4 The SOAP client-processor forwards the textual XML format to the HTTP client running under CICS.

5 The HTTP client sends the SOAP envelope (in textual XML format) to a SOAP server (for example a platform using Microsoft .NET, IBM Websphere, Apache SOAP, or AXIS) via HTTP. This can also be routed via a SOCKS or Proxy server.

The reply is then sent from the SOAP server back to the CICS User Transaction (the SOAP client), using the reverse of the above steps (that is, steps 6 to 10). The SOAP server sends the reply back to the HTTP client, which forwards it to the SOAP client-processor. The SOAP client-processor calls the XML parser to parse the textual XML format into a tree representation. The tree is passed to the SOAP converter, which de-serializes the parameters into a binary representation, and forwards them to the CICS User Transaction. The CICS User Transaction then processes the reply.

Using Web Service Security Features When z/VSE Acts As the SOAP Client

These are the areas you should consider:

- *Transport-Layer Encryption*. When transport-layer authentication is used, z/VSE acts as an HTTP client. The HTTP client that is implemented in z/VSE then supports HTTPS. To use HTTPS:
  - The URL needs to specify `https://`.
  - You must provide a public/private key pair, together with certificates. For details of how to specify the keys, refer to the skeleton SKSOAPOP in VSE/ICCF Library 59.
- *Transport-Layer Authentication*. From z/VSE 4.2 onwards, the HTTP Client supports SSL/HTTPS. Therefore, you can use SSL client authentication using certificates. If requested, the SSL protocol can send the client's certificate to the server (service provider). If required, the server can use the client's certificate to perform authentication and authorization. In addition, HTTP basic authentication is supported by the z/VSE HTTP Client.
- *Message-Layer Authentication*. From z/VSE 4.2 onwards, the z/VSE SOAP Engine (that is, the z/VSE SOAP Client) supports the authentication token in the SOAP header. For the `UsernameToken`, the user application (or converter code) requesting the service must pass the `username` and `password` to the SOAP Engine. If authentication is done using a certificate, the certificate name must be provided. Code for passing this information can either be part of the user application, or part of the converter code.

How the IBM-Supplied SOAP Control Blocks Are Used

This topic describes the IBM-supplied SOAP control blocks, that are defined in the C-language header file IESSOAPH.H. You can find IESSOAPH.H in directory **...\<install-directory>\samples\soap\vseSoapClient**.

File IESSOAPH.H is used by all SOAP programs that run on the z/VSE host:

- The SOAP converter, SOAP server, and CICS User Transaction of Figure 156 on page 418.
- The SOAP converter, SOAP client-processor, and CICS User Transaction of Figure 157 on page 420.

Other control blocks, not described here, are used if you want to write your own SOAP converter (when the IBM-supplied converter does not meet your requirements). If you wish to write your own SOAP converter, refer to the header file documents for details of such control blocks.

How the SOAP_PARAM_HDR Control Block Is Used

The SOAP_PARAM_HDR control block is used to provide each parameter's data, consisting of a:
- name
- value (the data itself)
- length of the value
- type of value

as shown below.

```
char         name[16];      // parameter name
char         typename[16];  // data type name
unsigned int length;        // length of block (inc. header)
unsigned int type;          // type (see SOAP_TYPE_xxx)
```

Figure 158. Contents of the SOAP Parameter

1. A parameter's data is either:
   - passed to the CICS User Transaction,
   - generated by the CICS User Transaction.
2. The data is then converted by the SOAP converter either:
   - from native data (shown in Figure 158) to XML,
   - from XML to native data.

Here is a list of all possible values for the type of value field of Figure 158:

```
// Values for type field in SOAP_PARAM_HDR
#define SOAP_TYPE_UNSPECIFIED   0     // unknown/unspecified type
#define SOAP_TYPE_PRIVATE       1     // private type
#define SOAP_TYPE_STRUCT        2     // hirachical structure
#define SOAP_TYPE_ARRAY         3     // hirachical array structure
#define SOAP_TYPE_STRING        10    // String
#define SOAP_TYPE_INTEGER       11    // Integer (4 bytes)
#define SOAP_TYPE_SHORT         12    // Short (2 bytes)
#define SOAP_TYPE_BYTE          13    // Byte (1 byte)
#define SOAP_TYPE_BOOLEAN       14    // Boolean (1 byte)
#define SOAP_TYPE_BINARY        15    // Binary
#define SOAP_TYPE_LONG          16    // Long (8 bytes)
#define SOAP_TYPE_UINTEGER      17    // Unsigned Integer (4 bytes)
#define SOAP_TYPE_USHORT        18    // Unsigned Short (2 bytes)
#define SOAP_TYPE_UBYTE         19    // Unsigned Byte (1 byte)
#define SOAP_TYPE_ULONG         20    // Unsigned Long (8 bytes)
// Decimal data types (represented as Packed Decimal COMP-3)
#define SOAP_TYPE_DECIMAL       21    // Decimal (with decimal places)
#define SOAP_TYPE_DECINT        22    // Integer
// Date and Time specific types
#define SOAP_TYPE_DATETIME     100    // Date and Time in ISO 8601
#define SOAP_TYPE_DATE         101    // Date  (e.g. "2002-10-10")
#define SOAP_TYPE_TIME         102    // Time  (e.g. "13:20:00")
#define SOAP_TYPE_GYEAR        103    // Year  (e.g. "2005")
#define SOAP_TYPE_GMONTH       104    // Month (e.g. "--05")
#define SOAP_TYPE_GDAY         105    // Day   (e.g. "---01")
#define SOAP_TYPE_GYEARMONTH   106    // Year and Month (e.g. "1999-05")
#define SOAP_TYPE_GMONTHDAY    107    // Month and Day  (e.g. "--05-01")
```

Figure 159. Possible Values for `type` of Value Field

The entries in Figure 159 are self-explanatory, except for the three fields described below.

Type	Description
SOAP_TYPE_UNSPECIFIED	If the `type` is SOAP_TYPE_UNSPECIFIED, then the SOAP converter found an unknown type *without* a given namespace URL. In this case the data is not converted and is given as plain text. Your program must therefore check if it recognizes the type (even without the namespace), and must then convert the data itself.
SOAP_TYPE_PRIVATE	If the `type` is SOAP_TYPE_PRIVATE, then the SOAP converter found an unknown type which had a namespace URL. In this case, the data is not converted and is given as plain text. Your program must then check if it knows the namespace URL/type pair and convert the data itself. The namespace URL is contained in the SOAP_PROG_PARAM control block.
SOAP_TYPE_STRUCT	If the `type` is SOAP_TYPE_STRUCT, then the SOAP converter received a hierarchical structure. This might be a type of array, or even an array that is a member of an enclosing array (the depth of the hierarchy is unlimited). An example of a hierarchical structure might be: +------------+ | Header | +------------+ | +--------+
	Header	
+--------+
	Data	
+--------+
+--------+
	Header	
+--------+
	Data	
+--------+
...	 +------------+ The “outer” header contains the length of this header's data. The type is SOAP_TYPE_STRUCT. In this example, the data of the “outer” header contains two parameters (each consisting of a header and data). Each parameter can be of any length. The program can recognize if there are further parameters by checking the length of the outer header's data block against the current parameter length.
SOAP_TYPE_ARRAY	Arrays are handled similar to SOAP_TYPE_STRUCT. The number of items in an array is specified in high-order halfword of field `type` in SOAP_PARAM_HDR. The field `typename` specifies the `typename` of the inner parameters (array items).
SOAP_TYPE_DECIMAL	Decimal numbers are represented as packed decimal (COMP-3 in COBOL). The implied decimal position (number of decimal digits) is specified in high-order halfword of field `type` in SOAP_PARAM_HDR.

How the SOAP_PROG_PARAM Control Block Is Used

The SOAP_PROG_PARAM control block is passed to a CICS User Transaction when the transaction is called as a *SOAP service* (for example in Figure 156 on page 418 the CICS User Transaction running on the host would receive SOAP_PROG_PARAM at Step 5).

The CICS User Transaction is called by the converter program (for example IBM-supplied IESSOAPE). It contains the requested SOAP method name, the names of the input and output queue to use and maybe a namespace URL for the contained parameter types (see SOAP_PARAM_HDR above). If you want to use a private type (SOAP_TYPE_PRIVATE) as a parameter to return, you can specify your own namespace URL here. The converter will build the XML using your namespace/type pair.

Here is a list of fields contained in the SOAP_PROG_PARAM control block:

```
char  method[16];        // (in)     method name
char  inqueue[8];        // (in)     input params
char  outqueue[8];       // (in)     output params
char  namespaceurl[128]; // (in/out) private namespace url
int   retcode;           // (out)    return code
char  methodlong[128];   // (in)     long method name
// Authentication
int   authtype;          // (in) Authetication type (AUTH_xxx)
char  authuser[64];      // (in) user id for authentication
char  authpwd[64];       // (in) password for authentication

// Authentication types
#define AUTH_NONE              0  // No authentication
#define AUTH_HTTP_BASIC        1  // basic HTTP authentication
#define AUTH_WS_PLAIN          2  // authentication using plain pwd
#define AUTH_WS_DIGEST         3  // authentication using digest
#define AUTH_WS_CERT           4  // authentication using certificate
#define AUTH_SSL_CERT          5  // SSL client authentication
```

Figure 160. Fields Contained in SOAP_PROG_PARAM Control Block

Notes:

1. The methodlong field supports method names that are greater than 16 chars.
2. For compatibility with earlier releases, the method field contains the first 16 characters of the method name.
3. For AUTH_HTTP_BASIC and AUTH_WS_PLAIN, the fields authuser and authpwd specify the user-ID and password that were received from the client-side using the specified method. For AUTH_WS_DIGEST, field authuser contains the user-ID, and authpwd specifies the name of a TS-QUEUE that holds these three entries:
   a. The password digest in base64.
   b. The created timestamp as text.
   c. The nonce value in base64.
4. For AUTH_WS_CERT and AUTH_SSL_CERT the field authuser contains the user-ID that was mapped for the certificate or blanks if no user was mapped. The field authpwd contains the name of a TS-QUEUE that holds one entry:
   a. The binary certificate data.

How the SOAP_DEC_PARAM Control Block Is Used

The SOAP_DEC_PARAM control block must be used by a CICS User Transaction acting as a SOAP client, to call the SOAP converter. The SOAP converter then calls the SOAP client-processor, which performs the SOAP call to the SOAP server. For details, see Figure 157 on page 420.

If the field proxytype is not HTTP_TYPE_DIRECT (0), you have to fill in the required proxy fields for this type.

```
char  url[128];        // (in) the servers url
char  method[16];      // (in) method name
char  urn[128];        // (in) the urn
char  inqueue[8];      // (in) input queue name
char  outqueue[8];     // (in) output queue name
char  namespaceurl[128]; // (in/out) namespace url
// proxy
int   proxytype;       // (in) proxy type (HTTP_TYPE_xxx)
char  proxy[128];      // (in) proxy server
int   proxyport;       // (in) port number
char  userid[16];      // (in) userid for socks
char  password[16];    // (in) password for socks 5
// return code
int   retcode;         // (out) return code
// SOAP Action
char  soapaction[128]; // (in) SOAPAction value
char  methodlong[128]; // (in) long method name
// Authentication
int   authtype;       // (in) Authetication type (AUTH_xxx)
char  authuser[64];    // (in) user id for authentication
char  authpwd[64];     // (in) password for authentication

// Authentication types
#define AUTH_NONE              0  // No authentication
#define AUTH_HTTP_BASIC        1  // basic HTTP authentication
#define AUTH_WS_PLAIN          2  // authentication using plain pwd
#define AUTH_WS_DIGEST         3  // authentication using digest
#define AUTH_WS_CERT           4  // authentication using certificate
#define AUTH_SSL_CERT          5  // SSL client authentication
```

Figure 161. Fields Contained in SOAP_DEC_PARAM Control Block

Notes:

1. The methodlong field is used for supporting method names that are greater than 16 characters.
2. If the methodlong field contains all zeros or blanks, or if the COMMAREA length that is passed indicates that the new field is not present, the method name is taken from the previously-used field method. This provides backward compatibility with existing programs.
3. If the COMMAREA is large enough and methodlong is not zero or blanks, the method name is taken from methodlong.
4. For AUTH_HTTP_BASIC, AUTH_WS_PLAIN and AUTH_WS_DIGEST, the fields authuser and authpwd specify the user-ID and password to be sent to the server using the chosen method. For AUTH_WS_CERT, the field authuser specifies the name of the certificate member to use, for example CRYPTO.KEYRING(CERTNAME). In this case, field authpwd is not used and is ignored. AUTH_SSL_CERT is not supported for SOAP client operations. However, the server-side can decide to use the client certificate that is used with SSL and perform client authentication.
5. If the COMMAREA length that is passed indicates that the fields authtype, authuser and authpwd are not present, *no authentication* will be used (backward compatibility).

These are the defined proxy types you can use if no direct connection to the SOAP server is available:

```
// Proxy types
#define HTTP_TYPE_DIRECT      0 // direct connection
#define HTTP_TYPE_PROXY       1 // connection through a proxy
#define HTTP_TYPE_SOCKS4      2 // connection through Socks V4
#define HTTP_TYPE_SOCKS5      3 // connection through Socks V5
```

Figure 162. Proxy Types That Can Be Used With SOAP_DEC_PARAM Control Block

Configuring the z/VSE SOAP Engine

The z/VSE SOAP Engine is shipped by IBM as pre-installed and pre-configured. You are normally not required to change the configuration. Here is an example of a SOAP Engine configuration:

```
* $$ JOB JNM=SOAPOPT,CLASS=A,DISP=D
// JOB SOAPOPT GENERATE SOAP OPTION PHASE
* *******************************************************************
* ASSEMBLE AND LINK THE SOAP OPTION PHASE                           *
* *******************************************************************
// LIBDEF *,CATALOG=PRD2.CONFIG
// LIBDEF *,SEARCH=PRD1.BASE
// OPTION ERRS,SXREF,SYM,NODECK,CATAL,LISTX
   PHASE IESSOAPO,*,SVA
// EXEC ASMA90,SIZE=(ASMA90,64K),PARM='EXIT(LIBEXIT(EDECKXIT)),SIZE(MAXC
               -200K,ABOVE)'
IESSOAPO CSECT
IESSOAPO AMODE ANY
IESSOAPO RMODE ANY
* ******************************************************************
* General settings.
* ******************************************************************
*
* TRACE FLAGS:   USED TO ACTIVATE TRACING
*
TRSYSLOG DC     XL2'001F'   TRACE TO SYSLOG
TRSYSLST DC     XL2'001F'   TRACE TO SYSLST
TRC_SERV EQU    X'0001'
TRC_CLNT EQU    X'0002'
TRC_DEC  EQU    X'0004'
TRC_ENC  EQU    X'0008'
TRC_HTTP EQU    X'0010'
*
* ******************************************************************
* Codepage specific settings.
* ******************************************************************
*
* SOAP SERVER EBCDIC CODEPAGE (IBM CCSID format)
* POSSIBLE VALUES ARE DESCRIBED IN THE MANUAL
* CICS Family: Communicating from CICS on System/390.
* See chapter Charater data
*
SERV_ECP DC     CL8'037'
*
* SOAP SERVER ASCII CODEPAGE
* POSSIBLE VALUES ARE DESCRIBED IN THE MANUAL (HTML coded format)
* CICS Transaction Server for VSE/ESA - Internet Guide
* See appendix: HTML-coded character sets
*
SERV_ACP DC     CL40'iso-8859-1'
*
*
* SOAP CLIENT EBCDIC CODEPAGE (ICONV FORMAT)
* POSSIBLE VALUES ARE DESCRIBED IN THE MANUAL
* LE/VSE C Run-Time Programming Guide
* See chapter Internationalization
*
```

```
CLNT_ECP DC     CL16'IBM-1047'
*
* SOAP CLIENT ASCII CODEPAGE (ICONV FORMAT)
* POSSIBLE VALUES ARE DESCRIBED IN THE MANUAL
* LE/VSE C Run-Time Programming Guide
* See chapter Internationalization
*
CLNT_ACP DC     CL16'ISO8859-1'
*
* ******************************************************************
* SSL specific settings. Used for VSE as SOAP client when the URL
* starts with https:// (secure HTTP).
* ******************************************************************
*
* Keyring library for SSL
*
SSLKEYR  DC     CL16'CRYPTO.KEYRING'
*
* Keyname for SSL
*
SSLKEYNM DC     CL8'HTTPSKEY'
*
* Ciper Specs for SSL
*
SSLCIPSP DC     CL64'010208090A62'
*
* Session timeout for SSL
*
SSLTIMEO DC     F'86440'
*
* ******************************************************************
* Parameter name mapping (short names to long names):
* When receiving SOAP messages with paramater names that are longer
* than 16 chars (and would not fit into the SOAP_PROG_PARAM header),
* the following table is searched andte long name is replaced by the
* corresponding short name.
* When sending out SOAP messages, the names of all parameters are
* checked against the table, if the short name stats with the '#'
* character. If a matching short name is found, the parameter name
* is replaced by the corresponding long name.
* ******************************************************************
*
* Address of mapping table (see below)
*
PNAMETAB DC     A(TABSTART)
*
* Short name indication (usually '#') for outgoing parameters
*
SNAMEIND DC     CL1'#'
*
* ******************************************************************
* Macro definition used for mapping table below.
* ******************************************************************
         MACRO
&NAME    ENTRY &SHORT=,&LONG=
&NAME    DC    Y(N&SYSNDX*)     LENGTH OF THIS ENTRY
         DC    Y(L'S&SYSNDX)    LENGTH OF SHORT NAME
         DC    Y(L'L&SYSNDX)    LENGTH OF LONG NAME
S&SYSNDX DC    C&SHORT          SHORT NAME
L&SYSNDX DC    C&LONG           LONG NAME
N&SYSNDX DC    0H               START OF NEXT ENTRY
         MEND
*
* ******************************************************************
* Start of the mapping table. Entries are specified as follows:
*      ENTRY SHORT='short name',LONG='long name'
* ******************************************************************
*
TABSTART DS     0F              START OF TABLE
*
          ENTRY SHORT='#shortin',                                X
```

```
               LONG='ThisIsAVeryLongInputWithMoreThan16Chars'
          ENTRY SHORT='#shortout',                                    X
               LONG='ThisIsAVeryLongOutputWithMoreThan16Chars'
*
TABEND   DC     H'0'            END OF TABLE
*
         END
/*
// IF $MRC GT 4 THEN
// GOTO NOLINK
// EXEC LNKEDT,PARM='MSHP'
* *******************************************************************
* DONT FORGET TO NEWCOPY THE IESSOAPO IN ORDER TO ACTIVATE IT:      *
*   CEMT SET PROG(IESSOAPO) NEWCOPY                                 *
* *******************************************************************
/. NOLINK
/*
/&
* $$ EOJ
```

However, if you have specific requirements (for example, you wish to use a different code page), you should use the skeleton SKSOAPOP in VSE/ICCF Library 59 to generate the SOAP option phase **IESSOAPO**.

You can tailor the follow settings within the SOAP configuration:
- The code pages used for ASCII/EBCDIC translation
- The settings for SSL/HTTPS.
- The long-to-short name mapping for SOAP parameter names.

Mapping Long-Names to Short-Names

Due to the size restriction for TS Queue entries, SOAP parameters can only have names up to 16 characters (as shown in the `SOAP_PROG_PARAM` control block).

If you wish to use SOAP parameters that are greater than 16 characters, you can supply your own mapping to map long names (greater than 16 chars) to short names (less than or equal to 16 characters). The z/VSE SOAP Engine will translate:
- Long names to their corresponding short names when it receives SOAP messages that contain parameters with long names.
- Short names to their corresponding long names when sending out SOAP messages containing parameters with long names.

Short names that belong to a long name must start with a "**#**" character, so that the z/VSE SOAP Engine can recognize this as a name that needs to be translated.

Description of the IBM-Supplied SOAP Service (getquote.c)

The source code for the IBM-supplied SOAP service is `getquote.c`. You can find this C for CICS program in directory **...\<install-directory>\samples\soap\vseSoapService**. Also in this directory you will find the jobs to compile and link `getquote.c` (jobs `compile.job` and `link.job`).

1. SOAP Server Calls SOAP Service: The program `getquote` is invoked by the SOAP server (via the SOAP converter), as shown in Step 5 of Figure 156 on page 418. Each request contains the:
- TargetObjectURI, which defines which object is to be called for this request
- method name of the method to call in this object.

The URI (Uniform Resource Identifier) used for `getquote` is `urn:iessoapd:getquote`. It always starts with `urn`, followed by the name of the:
1. SOAP converter (the IBM-supplied `iessoapd`)
2. SOAP service (the IBM-supplied C for CICS program `getquote`).

You could use your own SOAP converter instead of `iessoapd`. The method name for the IBM-supplied sample is `getQuote`, which is the name given to the SOAP service. The SOAP service uses the method name to decide which operation is do be done. Therefore each SOAP service (or CICS program) can handle several methods.

2. Map COMMAREA to SOAP_PROG_PARAM: When the SOAP service is started, it must retrieve the pointer to the provided CICS COMMAREA. The COMMAREA must be mapped to the C structure SOAP_PROG_PARAM, as shown in Figure 163.

```
SOAP_PROG_PARAM* call;
EXEC CICS ADDRESS COMMAREA(call);
```

Figure 163. Mapping COMMAREA to SOAP_PROG_PARAM Control Block

3. Check Which Method is Requested: The SOAP service can now check for the method that has been requested, as shown in Figure 164. The IBM-supplied SOAP service knows of only one method, `getQuote`: all other method names will produce an error code of 1. If the correct method name is requested, the SOAP service calls a sub-function, and provides two CICS queues that were built by the SOAP server to:
1. pass the given SOAP parameters to the SOAP service (via `inqueue`)
2. receive the output parameter from the SOAP service (via `outqueue`).

```
// check if the SOAP method name is 'getQuote'
if(strncmp(call->method, "getQuote", 8) == 0 )
    rc = ProcessGetQuote(call->inqueue, call->outqueue);
else
    rc = 1;
```

Figure 164. Checking Which SOAP Method Has Been Requested

4. Get Input Parameters: The SOAP service gets the input parameters sequentially by reading them from the CICS `inqueue`. The SOAP service then maps the value that was read to the SOAP_PARAM_HEADER structure, as shown in Figure 165. `pName` is the element name of the parameter, `pPtr` is a pointer to the data, and `pLen` is the length of the data.

```
SOAP_PARAM_HDR* param;
unsigned short length;
EXEC CICS READQ TS QUEUE(inqueue)
    SET(param) LENGTH(len) NEXT
    RESP(resp) RESP2(resp2);
if(param->type != SOAP_TYPE_STRING)
    return 5; // invalid type (for this service)
*pName = (char*)&param>name;
*pPtr  = (char*)&param[1];
*pLen  = param->length - sizeof(SOAP_PARAM_HDR);
```

Figure 165. Get Input Parameters from CICS Queue

5. Put Value into CICS Output Queue: To put a parameter into the CICS `outqueue` the SOAP service then copies the parameter into a SOAP_PARAM_HEADER structure, and writes this structure into the CICS `outqueue` , as shown in Figure 166 on page 430

on page 430.

```
int SetNextOutParameter(char*outqueue,
        char *typename, unsigned int type,
        char* name, char* ptr,int len)
{
  SOAP_PARAM_HDR* param;
  int resp,resp2;
  param = (SOAP_PARAM_HDR*)malloc(sizeof(SOAP_PARAM_HDR)+len);
  if(param==NULL)
      return(-1);
  memset(param->name,' ',sizeof(param->name));
  memcpy(param->name,name,strlen(name));
  memset(param->typename,' ',sizeof(param->typename));
  memcpy(param->typename,typename,strlen(typename));
  param->type = type;
  param->length = len + sizeof(SOAP_PARAM_HDR);
  memcpy(&param[1],ptr,len);
  EXEC CICS WRITEQ TS QUEUE(outqueue)
            FROM(param) LENGTH(len+sizeof(SOAP_PARAM_HDR))
            RESP(resp) RESP2(resp2);
  free(param);
  return(0);
};
```

Figure 166. Put Parameter Into the CICS Output Queue

Remaining Processing: The remaining processing is fairly uncomplicated:

1. The SOAP service reads the first parameter from the CICS `inqueue`, and checks if the:
   - parameters name is `symbol`
   - type is `SOAP_TYPE_STRING`.

   If not, the SOAP service returns an error code to the SOAP server.
2. For illustration purposes, the SOAP service uses a hard-coded symbol value, and puts this value (as a parameter) into the CICS `outqueue` together with the name `data` and `SOAP_TYPE_STRING`.
3. The SOAP service should now exit, returning an error code of zero to the SOAP server (via the SOAP converter). The SOAP server creates the XML representation of the returned value in the CICS `outqueue` and sends it back to the SOAP client (see Steps 9 and 10 of Figure 156 on page 418).

Description of the IBM-Supplied SOAP Client (soapclnt.c)

The source code for the IBM-supplied SOAP client is `soapclnt.c`. You can find this C for CICS program in directory **...\<install-directory>\samples\soap\vseSoapClient**. Also in this directory you will find jobs to compile and link `soapclnt.c` (jobs `compile.job` and `link.job`).

1. Preparing to Call the SOAP Service: Program `soapclnt.c` calls the SOAP service residing on the z/VSE host, as shown in Steps 1 to 5 of Figure 157 on page 420.

Each call contains the:
- URL (Uniform Resource Locator) of the SOAP server
- name of the program that is to be called by the SOAP server (the URI)
- method to be requested from the called program (METHOD).

An example is shown in Figure 167 on page 431.

```
char *URL = "http://9.164.155.95:1080/cics/CWBA/IESSOAPS";
char *URN = "urn:iessoapd:getquote";
char *METHOD = "getQuote";
```

Figure 167. Preparing the SOAP Client's Call Parameter

2. Prepare the SOAP_DEC_PARAM Structure: The SOAP client prepares a structure of type SOAP_DEC_PARAM by copying the values required for the call into the SOAP_DEC_PARAM structure, as shown in Figure 168.

```
// prepare the call
strncpy(call.url, URL, strlen(URL));
strncpy(call.method, METHOD, strlen(METHOD));
strncpy(call.urn, URN, strlen(URN));
strncpy(call.inqueue, "INPUT   ",8);
strncpy(call.outqueue,"OUTPUT  ",8);
call.proxytype = 0;
call.proxytype = HTTP_TYPE_DIRECT;
call.authtype  = AUTH_NONE;
```

Figure 168. SOAP Client Prepares the SOAP_DEC_PARAM Structure

3. Insert Parameters into CICS Input Queue of the SOAP Server: The SOAP client uses the CICS `inqueue` and `outqueue` (as does the SOAP service) to transmit parameters. In the IBM-supplied example, the SOAP client requests the stock quote for symbol `IBM`. In response, a parameter called `symbol`, with value `IBM`, and type `SOAP_TYPE_STRING`, is inserted into the CICS `inqueue` as shown in Figure 169.

```
SetNextOutParameter("INPUT   ",
    "string", SOAP_TYPE_STRING,
    "symbol", "IBM", 3);
```

Figure 169. SOAP Client Inserts Values into the SOAP Server's Input Queue

4. Call SOAP Converter to Handle Requests: Now that the preparation of the call is complete, the SOAP client can call the SOAP converter (IESSOAPE) to handle the SOAP client requests, as shown in Figure 170. The SOAP converter in turn calls the SOAP client-processor (as shown in Step 2 of Figure 157 on page 420).

```
EXEC CICS LINK PROGRAM("IESSOAPE")
    COMMAREA(&call) LENGTH(sizeof(call))
    RESP(rc) RESP2(rc2);
```

Figure 170. SOAP Client Calls SOAP Converter (IESSOAPE) to Handle Requests

5. Obtain Results of the SOAP Call: After the SOAP client has called the SOAP converter (IESSOAPE), it can then obtain the result of this call, as shown in Figure 171. The variable *call.namespaceurl* contains the namespace URL that was used by the SOAP service that was called, to encode the content of the reply. If *namespace url* contains a value, then the URL was *not* know to the decoder. The SOAP client must therefore deserialize the value of the parameter itself.

```
rc = GetNextInParameter("OUTPUT  ",&name,&val,&len);
if(rc!=0)
    break;
cicsprintf(  "name/type = %.32s", name );
cicsprintf(  "len       = %d", len);
sprintf(temp,"val       = %s%ds", "%.", len);
cicsprintf(temp,val);
```

Figure 171. SOAP Client Obtains Results of the SOAP Call

6. Delete CICS Queues That Were Created: After the SOAP call is completed, the SOAP client must now delete the temporary CICS queues that were automatically created during the preparation of the call. This ensures that all memory is given back to CICS. This is shown in Figure 172.

```
EXEC CICS DELETEQ TS QUEUE("INPUT   ")
    RESP(rc) RESP2(rc2);
EXEC CICS DELETEQ TS QUEUE("OUTPUT  ")
    RESP(rc) RESP2(rc2);
```

Figure 172. SOAP Client Deletes CICS Queues

Using a Java SOAP Client

Instead of using the IBM-supplied SOAP client written in C for CICS (`soapclnt.c`), you can use a Java SOAP client.

The SOAP client `GetQuote.java` is supplied with the Apache SOAP distribution. This example is stored in the Apache AXIS directory **axis-bin-1_4\axis-1_4\samples\stock\GetQuote.java**, contained in the Apache SOAP package (see "Step 1: Download and Install the Java SOAP Client Packages on the Client" on page 433 for details).

You can also find an IBM-modified version of `GetQuote.java` in the directory **...\<install-directory>\samples\soap\javasample**, together with batch files to compile and run the program. The sample has been modified so that it calls the IBM-supplied `getQuote` SOAP service, as shown by the highlighted statement of Figure 173.

```
...
Call     call    = (Call) service.createCall();
call.setTargetEndpointAddress( url );
call.setOperationName( new QName("urn:iessoapd:getquote", "getQuote") );
call.addParameter( "symbol", XMLType.XSD_STRING, ParameterMode.IN );
call.setReturnType( XMLType.XSD_STRING );
Object ret = call.invoke( new Object[] {symbol} );
...
```

Figure 173. SOAP Client Calls the getQuote Service

As you can see, the setup of the call is similar to the setup when a SOAP client runs on the z/VSE host (as shown in Figure 168 on page 431). The Java client sample performs the same processing as the SOAP client sample (`soapclnt`) that runs on the z/VSE host.

You must change the `run.bat` file (contained in the directory **...\<install-directory>\samples\soap\javasample**) so that the URL of the SOAP server and the requested symbol, are given to the program on the command line. For details, refer to "Running the IBM-Supplied SOAP Sample" on page 433.

Running the IBM-Supplied SOAP Sample

The IBM-supplied SOAP sample consists of three programs:

- GetQuote.java (which is a SOAP client running on a Java-enabled platform)
- getquote.c (which implements a SOAP service on the z/VSE host)
- soapclnt.c (which is used when the z/VSE host acts as a SOAP client)

The program getquote.c can be called as a SOAP service from either GetQuote.java or soapclnt.c . The IBM-supplied SOAP sample illustrates both of these scenarios, where the z/VSE host acts as:

- a SOAP client (when soapclnt.c is the SOAP client)
- a SOAP server (when GetQuote.java is the SOAP client).

To run the IBM-supplied SOAP sample, you must follow the steps described in this topic. However, before you can run this SOAP sample you must have:

- Installed the Java Development Kit (JDK) 1.5 or later. If you do not have JDK 1.5 or later installed, refer to "Installing and Configuring Java" on page 19 for details of how to install it.
- The IBM *C-compiler for VSE/ESA* installed.

Step 1: Download and Install the Java SOAP Client Packages on the Client

You must download various packages, if you do not already have the required files installed. The package you require is the Apache AXIS package, which you can obtain from this URL:

```
http://ws.apache.org/axis/
```

At this Web site, go to the download directory containing the latest version, and download the axis-bin package (for example, **axis-bin-1_4.zip**).

Step 2: Extract and Install the Required Java Programs

Step 2.1: Create a New Directory

In this step, you create a new directory. To simplify the CLASSPATH definition, all .JAR files required for running the SOAP sample will be saved into this directory. In the steps below, this directory is called **work**.

Step 2.2: Extract JAR Files and Place in Your Directory

In this step you extract the required **.JAR** files from the downloaded ZIP file **axis-bin-1_4.zip**, and put the extracted JAR files into your directory **work**:

- axis.jar
- commons-discovery-0.2.jar
- commons-logging-1.0.4.jar
- jaxrpc.jar
- log4j-1.2.8.jar
- saaj.jar
- wsdl4j-1.5.1.jar

Step 3: Compile /Link the Sample C Programs, and Define Them to CICS

Before you can start to use the IBM-supplied C-for-CICS programs (the SOAP client and the SOAP service), you must compile and link the source for these programs, and then define the programs to CICS. You should therefore:

1. change the URL in the SOAP client (soapclnt.c) to match your z/VSE system. The sample URL is as follows:

   ```
   http://9.164.123.23:1080/cics/CWBA/IESSOAPS
   ```

   (Usually, you simply need to change the IP-address).
2. upload the source for:
   - soapclnt.c from directory **...\<install-directory>\samples\soap\vseSoapClient**
   - getquote.c from directory **...\<install-directory>\samples\soap\vseSoapService**

   to a z/VSE library. Use either the TCP/IP **ftp** utility, or the File Transfer function provided with a 3270 Emulator.
3. compile source programs soapclnt.c and getquote.c using the IBM-supplied jobs `compile.job` and `link.job`. which you can find in the same directories as the relevant source programs (soapclnt.c or getquote.c). You might need to adapt these jobs to your own environment (source and destination library names, and so on). The compile Job performs the following processing:
   a. Invokes the CICS pre-compiler.
   b. Invokes the C compiler.
   c. Catalogs the object deck into the z/VSE library that you specify in your compile job.

   The link Job performs the following processing:
   a. Invokes the LE pre-linker.
   b. Invokes the linkage editor.
   c. Catalogs the phase into the z/VSE library that you specify in your link job.
4. define the programs SOAPCLNT and GETQUOTE to CICS, using the CEDA transaction. You must also ensure that the phase you created above, can be located by CICS (using the LIBDEF statement of CICS). Figure 174 shows how you use CEDA for the GETQUOTE program.

```
CEDA DEFINE PROGRAM
OVERTYPE TO MODIFY
 CEDA  DEFine PROGram(         )
  PROGram      ==> GETQUOTE
  Group        ==> VSESPG
  DEscription  ==> SAMPLE SOAP SERVICE GETQUOTE
  Language     ==> C                 CObol | Assembler | C | Pli
  RELoad       ==> No                No | Yes
  RESident     ==> No                No | Yes
  USAge        ==> Normal            Normal | Transient
  USEsvacopy   ==> No                No | Yes
  Status       ==> Enabled           Enabled | Disabled
  RSl            : 00                0-24 | Public
  Cedf         ==> Yes               Yes | No
  DAtalocation ==> Any               Below | Any
  EXECKey      ==> User              User | Cics
 REMOTE ATTRIBUTES
  REMOTESystem ==>
  REMOTEName   ==>
  Transid      ==>
```

Figure 174. Using CEDA to Define Sample SOAP Service to CICS

Step 4: Configure CICS to Use CICS Web Support

You must now configure and activate CICS Web Support in order to use Web Services in z/VSE.

For details of how to do so, refer to "Configuring CICS Web support" in the *CICS Transaction Server for VSE/ESA, Internet Guide*, SC34-5765.

Step 5: Define the SOAP Server to CICS

To activate the IBM-supplied SOAP server, you must define a TCP/IP service for CICS. All SOAP clients (whether written in C-for-CICS or Java) require that you perform this step. Perform the CEDA DEFINE shown below.

```
CEDA   DEFine TCpipservice( SOAP     )
 TCpipservice   : SOAP
 Group          : VSESPG
 Description  ==> TCP/IP SERVICE FOR SOAP SEVRER
 Urm          ==> DFHWBADX
 Portnumber   ==> 01080             1-65535
 Certificate  ==>
 STatus       ==> Open              Open | Closed
 SSl          ==> No                Yes | No | Clientauth
 Attachsec    ==> Local             Local | Verify
 TRansaction  ==> CWXN
 Backlog      ==> 00001             0-32767
 TSqprefix    ==>
 Ipaddress    ==>
 SOcketclose  ==> No                No | 0-240000
```

Figure 175. Using CEDA to Define SOAP Server to CICS

Step 5: Activate the ASCII to EBCDIC Converter

An ASCII / EBCDIC converter (DFHCNV) is provided in VSE/ICCF library 59. It is used by CICS Web Support (and therefore by the SOAP server) to convert:

- Incoming XML data from ASCII to EBCDIC.
- Outgoing XML data from EBCDIC to ASCII.

To activate the ASCII / EBCDIC converter, submit skeleton DFHCNV to the VSE/POWER reader queue.

Step 6: Compile the Java Sample

1. Copy these files from the directory **...\<install-directory>\samples\soap\javasample** to your directory **work**:
   - compile.bat
   - run.bat
   - GetQuote.java
2. Run the file `compile.bat`.

Step 7: Run the Java SOAP Client Sample

1. Open the file `run.bat` using a text editor, and change the URL to match your own z/VSE system. The sample URL is as follows:

```
http://9.164.123.23:1080/cics/CWBA/IESSOAPS
```

   (Usually, you simply need to change the IP-address).
2. Start the `run.bat` file. A SOAP call is then sent from the SOAP client to the SOAP server on the z/VSE host, requesting the price for the stock symbol of IBM. The sample SOAP service running on the z/VSE host then returns a hard-coded symbol value to the Java program.
3. The Java SOAP client outputs the value of the stock (which is “hard-coded” in the program!) to the screen.

Step 8: Run the C-Program SOAP Client Sample

To run the C program soapclnt.c, you must:

1. define (again using CEDA) a CICS transaction that will call the CICS program SOAPCLNT that you defined in “Step 3: Compile /Link the Sample C Programs, and Define Them to CICS” on page 434. **Note:** ensure that you have changed the URL in the SOAP client (soapclnt.c) to match your z/VSE system!.
2. start the CICS transaction defined above. A SOAP call is then sent from the SOAP client to the SOAP server on the z/VSE host, requesting the price for the stock symbol of IBM. The sample SOAP service running on the z/VSE host then returns a hard-coded symbol value to the C program.

Writing Your Own SOAP Programs

The IBM-supplied sample which you can use as a template for writing your own programs, consists of:

- a SOAP C language header file `IESSOAPH.H` , which is described in “How the IBM-Supplied SOAP Control Blocks Are Used” on page 421. The COMMAREA and memory mappings used by the z/VSE SOAP implementation are defined in `IESSOAPH.H`. If you want to write SOAP services or clients in another language (such as COBOL) you must adopt these definitions to this language by yourself. A mapping for the assembler language is included in `IESSOAPH.H` as comments.
- a SOAP service `getquote.c` written in C for CICS, which is described in “Description of the IBM-Supplied SOAP Service (getquote.c)” on page 428
- a SOAP client `soapclnt.c` written in C for CICS, which is described in “Description of the IBM-Supplied SOAP Client (soapclnt.c)” on page 430.
- a SOAP client `GetQuote.java` written in Java, which is described in “Using a Java SOAP Client” on page 432.

If the SOAP converters IESSOAPD and/or IESSOAPE do not meet your requirements, you can replace them with your own programs. You specify the name of the SOAP converter for the z/VSE inbound direction in the URI of the SOAP request. For details, see “Description of the IBM-Supplied SOAP Service (getquote.c)” on page 428.

Chapter 25. Using the VSE Script Connector for Non-Java Access

This chapter describes how you use the *VSE Script connector* (which consists of the VSE Script Client and the VSE Script Server) to access z/VSE host or physical/logical middle-tier data from *non-Java* platforms.

Access from non-Java platforms is the main advantage of using the VSE Script connector. However, you can also use the VSE Script connector to access z/VSE host or physical/logical middle-tier data from *Java platforms*.

For each VSE Script Client that you write, you will probably also need to write a corresponding VSE Script (using the VSE Script Language) to meet your own specific requirements. However, new VSE Script Clients can use existing VSE Scripts to access z/VSE host or physical/logical middle-tier data.

This chapter contains these main topics:
- "How the VSE Script Connector Can Be Used"
- "Overview of the Protocol Used Between Client and Server" on page 439
- "Writing VSE Scripts Using the VSE Script Language" on page 440
- "Sample Files You Can Use for Writing VSE Script Clients" on page 445
- "Example of Writing a VSE Script Client (and Its VSE Script)" on page 445
- "Obtaining Data From the Middle-Tier Using VSE Script Clients" on page 455

Related Topics:
- "Overview of the VSE Script Connector" on page 39
- Chapter 7, "Installing the VSE Script Connector," on page 39

How the VSE Script Connector Can Be Used

You can use the VSE Script Connector to obtain data from either the:
- z/VSE host.
- Physical/logical middle-tier.

Figure 176 on page 438 show how a VSE Script is called by a VSE Script Client.

© Copyright IBM Corp. 2000, 2013

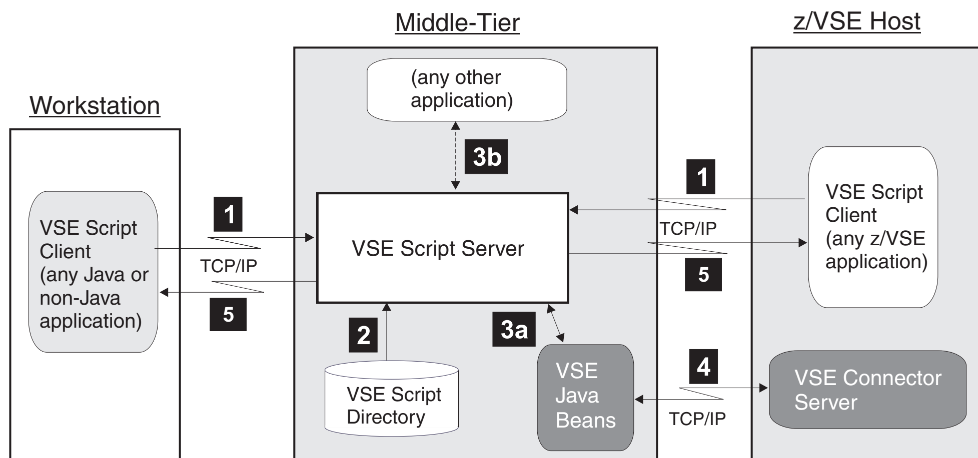


Figure 176. Obtaining Data From the Host or Middle-Tier Using the VSE Script Connector

1 The VSE Script Client establishes a TCP/IP connection to the VSE Script Server on the physical/logical middle-tier. The VSE Script Client then calls a VSE Script by sending:
- the file name of the VSE Script.
- the parameters belonging to the VSE Script

to the VSE Script Server running on a physical/logical middle-tier and Java-enabled platform.

2 The VSE Script Server reads the VSE Script file from the VSE Script directory, and starts to interpret and translate the VSE Script file statements. Each VSE Script file is written using the *VSE Script Language* (described in "Writing VSE Scripts Using the VSE Script Language" on page 440).

3a If the VSE Script file wishes to obtain data from the z/VSE host, the VSE Java Beans requests are executed by the VSE Java Beans.

3b The VSE Script file can also obtain data from any other application running on the physical/logical middle-tier, by calling the application. This is especially useful if you plan to run the VSE Script Client on the z/VSE host.

4 The VSE Java Beans communicate with the VSE Connector Server running on the z/VSE host, which performs the request for functions and data. (This step only applies when Step **3a** was followed).

5 The data that was obtained is then converted to a format that the VSE Script Client can use, and is returned to the VSE Script Client.

You can use the VSE Script Connector to execute *remote applications* (such as REXEC). For example:
- If you follow Step **3a**, VSE Script Clients can execute jobs on the z/VSE host.
- If you follow Step **3b**, VSE batch jobs or programs can execute applications or shell scripts on the physical/logical middle-tier.

Overview of the Protocol Used Between Client and Server

This topic describes the protocol used for the data flow between a VSE Script Client and the VSE Script Server that is installed on the physical/logical middle-tier. The general flow of actions is as follows:

1. The VSE Script Client opens a connection to the VSE Script Server.
2. Send a code page (**optional**):
   a. The first line (that is, the line before the script name is sent) is, for example, `**CODEPAGE=Cp1252**` followed by `CR LF (X'0D' X'0A')`. The code page line is always sent in UTF-8.
   b. Any following lines are sent either in the specified code page, or UTF-8 (if a code page is not specified).
3. The VSE Script Client then sends to the VSE Script Server:
   a. the name of a VSE Script, followed by a `CR LF (X'0D' X'0A')`
   b. each parameter value, followed by `CR LF`
4. After the last parameter has been sent, an empty line is sent (only `CR LF`). This indicates that all parameters have been transmitted.
5. The VSE Script Server starts to execute the VSE Script.
6. The VSE Script Server sends the output from the VSE Script back to the VSE Script Client, line-by-line. Each line is terminated by `CR LF`.
7. After it has sent the last output line to the VSE Script Client, the VSE Script Server then closes the connection.

You can easily implement the protocol described here into any kind of programming language. The programming language is only required to support calls to TCP/IP socket functions.

You can use a Telnet application to easily test the VSE Script Connector. To do so, you must:

1. Create a connection between the telnet client and the VSE Script Server.
2. Type the name of your VSE Script and press Enter to terminate the line with `CR LF`.
3. Type the name of each parameter and press Enter to terminate the line with `CR LF`.

   Note: You will not see what you type, since the VSE Script Server does not echo the data it receives.
4. After you have entered the last parameter, press Enter to start executing your VSE Script. The output from the VSE Script will now be displayed by the telnet client.

The example below shows a sequence where a VSE Script called `test.src` which contains three parameters, is executed. The parameters are `Hello`, `583`, and `get`. The VSE Script Server sends the output from the VSE Script (`Script test.src has been executed`) back to the VSE Script Client.

1. VSE Script Client to the VSE Script Server:

```
test.src<CR><LF>
Hello<CR><LF>
583<CR><LF>
get<CR><LF>
<CR><LF>
```

2. VSE Script Server to the VSE Script Client:

```
Script test.src has been executed<CR><LF>
```

Using SSL-Encrypted Connections

From z/VSE 4.2 onwards, the VSE Script Server supports *SSL-encrypted connections* for connections from VSE Script Clients to the VSE Script Server. If SSL is used, the protocol stays the same, but is instead sent over an encrypted SSL connection. To use SSL, you must have:

- Enabled SSL in TCP/IP for VSE/ESA.
- Created the required keys and certificates on:
  - The VSE Script Server-side.
  - z/VSE.

Writing VSE Scripts Using the VSE Script Language

The VSE Script connector includes a special programming language, called the *VSE Script Language*, that you can use to write your own VSE Scripts. These VSE Scripts are required for accessing VSE functions and data from your VSE Script Clients.

As with other programming languages, the VSE Script Language consists of:
- statements (if, while, for, break, continue, sub, gosub, return)
- operators (logical, concatenation, arithmetic)
- variables
- built-in functions.

The VSE Script Language is described in detail in the *Language Reference* online documentation, which is installed on your workstation during the installation of the VSE Script Server (as described in Chapter 7, "Installing the VSE Script Connector," on page 39.

This topic provides an introduction to the VSE Script Language in these sub-topics:
- "General Rules That Apply to the VSE Script Language"
- "VSE Script Language Built-In General Functions" on page 441
- "VSE Script Language: Built-In String Functions" on page 442
- "VSE Script Language: Built-In Console Functions" on page 443
- "VSE Script Language: Built-In POWER Functions" on page 443
- "VSE Script Language: Librarian Functions" on page 443
- "VSE Script Language: Built-In VSAM Functions" on page 444

General Rules That Apply to the VSE Script Language

Statements:

1. A statement can be either a:
   - keyword (IF, WHILE, ...)
   - function call
   - variable declaration
   - variable assignment.
2. Each statement must end with a semicolon (;), and any number of statements can be written on a single line.
3. A command cannot be split over two lines (statements on one line are considered as one command, including the `do` statement).
4. Blank lines are allowed.
5. The VSE Script Language is not case-sensitive.
6. Unrecoverable errors that occur when a VSE Script is interpreted are described as *ScriptErrors*. A ScriptError generates this information:
   - the line number and statement number where the error occurred.
   - the (unique) error number.

- a descriptive text for the error number.

Comments:

1. A comment begins with two forward slashes "//", and ends at the end of the line.
2. A comment can appear behind a `command`, or at the beginning of a line.

Variables:

1. An unlimited number of variables can be declared.
2. Each variable can be declared only once in a VSE Script
3. After they have been declared, all variables are global in the VSE Script.
4. A variable must be declared before use, and can be defined at any time/place.
5. Variable names must start with a alphabetic character. Afterwards, all alphabetic characters, digits and the underscore '_' are allowed.
6. There are three data types: int, string, and boolean.
7. Each variable can be accessed as an array of values.
8. `var[0]` is the same as the variable `var` itself, and therefore `var[0]=1` is the same as `var=1`.
9. The variable ARGV is always defined in a VSE Script, and a variable with this name cannot be declared.
10. The variable is of type STRING, the values of the variable are the parameters that are passed to the VSE Script during startup.

Librarian File Names:

1. A librarian *filename* is a string containing library, sublibrary, member and type, separated by "/", "\", or ".".
2. The format is either:
   - *library.sublib\member.type*
   - *library\sublib\member.type*
3. Librarian functions might expect to process:
   - A *library* only.
   - A *library* and a *sublib* only.
   - All the entries listed in (2.) above.

VSE Script Language Built-In General Functions

Function	Description
print / println	prints a comma separated list of values or expressions (int, string, boolean) to standard out
exit	exits the VSE Script with a specified return code
sleep	sends the VSE Script into sleep mode for a specified time interval
resetVar	deletes all values of a given variable, especially useful with arrays
arraySize	gets the actual size of a given array variable
getLastErrorMsg	gets the last internal error message. This message is set by many z/VSE access functions to give a detailed description of an error
readFile	reads a specified file line by line into a string array

Function	Description
saveFile	saves the lines in a string array to a file
getCallersIP	gets the IP address of the host that called the VSE Script. This could be used from inside a VSE Script to reject access to the VSE Script if other IP's than the expected try to call it.
call	calls an external program on the machine where the VSE Script Server runs.
callNoWait	calls an external program on the machine where the VSE Script Server runs. The function will return to the VSE Script immediately, it will not wait for the called program to end. Hence the output of the program can't be captured and no exit code is available.
VSEConnect	tries to connect to the specified host. You don't need to call this function explicitly, each function that access the host will automatically connect to VSE if needed. This function can be used to check if a connection is possible before any other operations start. Once established, the connection is pooled for later use.
putConsoleMsg	puts a console message to the z/VSE console. You should be aware that this is *not* a command!

VSE Script Language: Built-In String Functions

Function	Description
formatNumber	formats a given number to a string
toString	converts a variable (int, string, boolean) to a string
toInt	converts an integer or string to an integer.
indexOf	gets the index of the first occurrence of a given string in another string
lastIndexOf	gets the index of the last occurrence of a given string in another string
subString	gets a substring from a given string
trim	removes all leading and trailing blanks from a given string
strLen	gets the length of a string
readFileBinary	reads the specified binary file record- by- record into the array `destStr`. The `lrecl` value specifies the logical record length of a record. If `lrecl` is zero or not specified, a logical record length of 1024 is assumed. Each record is converted to its hexadecimal representation (for example X'01 A0 DD 02 00').
saveFileBinary	saves the records in specified string array `srcStr` into a specified binary file. Each record is converted from its hexadecimal representation into binary data.
strToBin	Converts the specified string into its binary representation using the code page `charset`, and stores the result.
binToStr	Converts the binary representation into its textual representation using the code page `charset`, and stores the result.

VSE Script Language: Built-In Console Functions

Function	Description
executeConsoleCmd	executes a console command and returns the result lines
putConsoleCmd	executes a console command. The output can be retrieved line by line using getConsoleMsg
getConsoleMsg	retrieves one line from the console
closeConsole	closes the console

VSE Script Language: Built-In POWER Functions

Function	Description
executePowerJob	executes a POWER job.
listPowerQueue	retrieves the member list of a POWER queue.
getPowerQueueEntry	receives a specified entry from a specified POWER queue.
getPowerQueueEntryASA	receives a specified entry from a specified POWER queue in ASA format.
putPowerQueueEntry	creates a new POWER queue member in the specified queue.
putPowerQueueEntryASA	creates a new POWER queue member in the specified queue in ASA format.
deletePowerQueueEntry	deletes a specified POWER queue entry.
cancelPowerQueueEntry	cancels a specified POWER queue entry.
releasePowerJob	releases a specified POWER queue entry in the reader queue.
getPowerEntryproperties	gets various properties of a POWER queue entry.
setPowerEntryProperties	sets various properties of a POWER queue entry.
getPowerQueueEntryBinary	has the same functionality and parameters as the getPowerQueueEntry function (described above), but receives the entry in BINARY format.
putPowerQueueEntryBinary	has the same functionality and parameters as the putPowerQueueEntry (described above), but sends the entry in BINARY format.

VSE Script Language: Librarian Functions

Function	Description
listLibraries	Retrieves a list of libraries, and stores it in the result array.
listSubLibs	Retrieves a list of sublibraries in the specified library, and stores it in the result array.
listMembers	Retrieves a list of members in the specified sublibrary, and stores it in the result array.
createSubLib	Creates a new sublibrary in the specified library.
deleteSubLib	Deletes a sublibrary in the specified library.
copyMember	Copies the *member source* to the *member target*.

Function	Description
moveMember	Moves the *member source* to the *member target*. Moving a member is the same as first performing a copyMember and then deleting the source member.
deleteMember	Deletes the specified member.
renameMember	Renames the specified member to a new *member name* and *member type*.
downloadMember	Downloads the specified member and stores its lines/records in the destination array.
uploadMember	Uploads the specified member by taking the lines/records from the `src` array.
getMemberFromPowerQueue	Copies the specified POWER queue entry into the specified member.
putMemberToPowerQueue	Copies the specified member to the specified POWER queue.

VSE Script Language: Built-In VSAM Functions

Function	Description
getVsamRecord	Retrieves a VSAM record from a given VSAM file
getVsamRecordGE	Retrieves a VSAM record from a VSAM file using the GreaterEqual search
getNextVsamRecord	Retrieves the next record based on given key values.
getPrevVsamRecord	Retrieves the previous record based on given key values
getVsamMappingInfo	Retrieves field informations for a given VSAM map or view.
getVsamMappingInfo2	Retrieves field informations for a given VSAM map or view. Has support for decimal positions.
deleteVsamRecord	Deletes a VSAM record from a VSAM file
insertVsamRecord	Inserts a new VSAM record into a VSAM file.
updateVsamRecord	Updates a VSAM record of a VSAM file
vsamBrowseStart	Starts a browse session for a given VSAM map or view. Only one session can be active for the same map or view. Afterwards you can browse through the returned records using vsamBrowseNext or vsamBrowsePrev. A call to vsamBrowseEnd ends the session and discards the internal browse data.
vsamBrowseNext	Can be called after a successful vsamBrowseStart. It positions to the next record of the retrieved records. The function can be called sequentially until the end of data is reached.
vsamBrowsePrev	Can be called after a successful vsamBrowseStart. It positions to the previous record of the retrieved records. The function can be called sequentially until the first record is reached.
vsamBrowseEnd	Ends the browse session for the specified filename.

Sample Files You Can Use for Writing VSE Script Clients

The files listed in Table 10 are copied to the directory where you installed the VSE Script Server (see "Step 1.2: Perform the Installation of the VSE Script Server" on page 41 for details):
- VSEScriptServlet.java
- VSEScriptWebService.java

You can use the files in Table 10 to write your own VSE Script Clients.

Table 10. Files Supplied for Writing VSE Script Clients

File	Description
VSEScriptClient.dll	Windows DLL for use, for example, with MS Office and Lotus Products
VSEScriptClient.h	C-Header for use with VSEScriptClient.dll
VSEScriptClient.lib	C-Library file for use with VSEScriptClient.dll
VSEScriptClient.123	Lotus 1-2-3 sample that uses VSEScriptClient.dll
VSEScriptClient.lss	Lotus Script source of the Visual Basic script
VSEScriptClient.xls	Microsoft Excel sample that uses VSEScriptClient.dll
VSEScriptClient.bas	Visual Basic source of the Visual Basic Script
VSEScriptCgi.c	Sample C-source of a CGI that uses VSEScriptClient.dll
VSEScriptCgi.exe	Compiled CGI
com\ibm\vse\script\client\ VSEScriptServlet.java	Java source of sample servlet
com\ibm\vse\script\client\ VSEScriptWebService.java	Java source of sample WebService

Example of Writing a VSE Script Client (and Its VSE Script)

This topic provides an example of how to write a VSE Script Client and its corresponding VSE Script, to insert VSAM data into office applications such as Lotus 1-2-3 or Lotus Wordpro.

The example uses a Windows DLL file **VSEScriptClient.dll**, which is available for use with any Windows programs (for example a Lotus Spreadsheet). You must, however, declare this DLL file using a statement like the one shown for Lotus 1-2-3 in Figure 180 on page 450. These statements are also contained in the online documentation, and you can easily Copy and Paste them as required.

This topic contains the following sub-topics:

- "Step 1: Setup the VSE Script Server Properties File" on page 446
- "Step 2: Setup the Connections Properties File" on page 446
- "Step 3: Define the Sample VSAM Data" on page 446
- "Step 4: Modify the Sample VSE Script" on page 447
- "Step 5: Start the VSE Connector Server on the z/VSE Host" on page 448
- "Step 6: Start the VSE Script Server Locally" on page 449
- "Step 7(a): Open the Sample Lotus 1-2-3 Spreadsheet File" on page 449
- "Step 7(b): Open the Sample MS Office Spreadsheet" on page 452
- "Step 7(c): Start a Sample VSE Script from the Command Line" on page 455

Other examples of how to program VSE Script Clients for other client platforms are provided in the online documentation supplied with the VSE Script Connector.

Step 1: Setup the VSE Script Server Properties File

The parameters for the VSE Script Server are defined in the file **VSEScriptServer.properties**, which you can find in the directory where you installed the VSE Script Server (see “Step 1.2: Perform the Installation of the VSE Script Server” on page 41 for details). An example of **VSEScriptServer.properties** is shown below.

```
#VSEScriptServer

#print messages (on) or do not print (off)
messages=on

#port where the server listens
listenport=4711

#number of maximum connections allowed
maxconnections=256

#root directory for VSE Scripts
scriptdirectory=./scripts

#name of the connection config file
connectionconfig=Connections.properties
```

The properties file is pre-customized for use with the sample, and you should not normally need to change this file. If you wish to do so, however, see “Step 2: Configure the VSEScriptServer Properties File” on page 42 for details.

Step 2: Setup the Connections Properties File

The **Connections.properties** file defines the settings for the z/VSE systems with which you wish to work. You can find **Connections.properties** in the directory where you installed the VSE Script Server. An example of this file is shown below.

```
connection.1.password=mypassw
connection.1.ip=9.12.34.56
connection.1.name=VSE01
connection.1.port=2893
connection.1.userid=jsch
connection.timeout=100
```

For details of the **Connections.properties**, see “Step 3: Configure the Connections Properties File” on page 43.

You can edit the above file using any text editor. Type your VSE password in clear text in the parameter `password`. When the VSE Script Server starts and reads the Connections properties file, the password is encrypted to parameter `encpassword`, and `password` is removed from the file. For details of the Connections properties file, see “Step 3: Configure the Connections Properties File” on page 43.

Step 3: Define the Sample VSAM Data

From VSE/ESA 2.5 onwards, a sample job SKVSSAMP is provided which is stored in ICCF library 59. It enables you to define sample data for various z/VSE e-business Connectors samples.

Submit the Job SKVSSAMP to define a VSAM cluster with the name VSAM.CONN.SAMPLE.DATA, in the VSESP.USER.CATALOG. It contains a number of records describing “used cars”. For details of job SKVSSAMP, see “4. Define the VSAM Data Cluster” on page 204. Job SKVSSAMP automatically uses the IDCAMS RECMAP command to define a VSAM map that contains the field

names of the sample data. For details, see “Defining a Map Using RECMAP” on page 105.

Step 4: Modify the Sample VSE Script

Figure 177 on page 448 shows the sample VSE Script **getdata.src**, which reads a record from the VSAM file defined in “Step 3: Define the Sample VSAM Data” on page 446, and inserts the data into the spreadsheet. You can find **getdata.src** in sub-directory **/scripts/samples** of the directory where you installed the VSE Script Server. The record is identified by its key, which is passed to the VSE Script by parameter. The VSE Script is used by the Lotus 1-2-3 example of a VSE Script Client, **VSEScriptClient.123**.

Make sure that you have the correct z/VSE host defined (using the constant **host**, shown as **VSE01** below). This host name *must* match the one defined in the **Connections.properties**.

```
// constants
String host      = "VSE01";
String file      = "VSESP.USER.CATALOG\\VSAM.CONN.SAMPLE.DATA\\USEDCARS";

// Variables
String keyfields;
String keyvalues;
String fields;
String values;
int rc;

// prepare the fields
keyfields[0] = "ARTICLENO";
keyvalues[0] = argv[0];   // argument is key

fields[0] = "ARTICLENO";
fields[1] = "MANUFACTURER";
fields[2] = "TYPE";
fields[3] = "MODEL";
fields[4] = "HP";
fields[5] = "DISPLACEMENT";
fields[6] = "CYLINDERS";
fields[7] = "COLOUR";
fields[8] = "FEATURES";
fields[9] = "PRICE";

// get the record
getVSAMRecord(host,file,&keyfields,&keyvalues,&fields,&values, &rc);

// print received data

if (rc!=0) do;
  println("Not found");
else do;
  println(values[0]);
  println(values[1]);
  println(values[2]);
  println(values[3]);
  println(values[4]);
  println(values[5]);
  println(values[6]);
  println(values[7]);
  println(values[8]);
  println(values[9]);
endif;
```

Figure 177. VSE Script Provided With the VSE Script Connector Example

Step 5: Start the VSE Connector Server on the z/VSE Host

Make sure that the VSE Connector Server is started in **non-SSL** mode. The server runs in class R by default. Use job STARTVCS, which is located in the reader queue, to start the server (for details see “Starting the VSE Connector Server” on page 34).

Step 6: Start the VSE Script Server Locally

To start the VSE Script Server locally, you use either:

- **runserver.bat** on Windows
- **runserver.cmd** on OS/2
- **runserver.sh** on Linux/Unix workstations.

Step 7(a): Open the Sample Lotus 1-2-3 Spreadsheet File

The sample Lotus Spreadsheet file **VSEScriptClient.123** already has the necessary setup to run the VSE Script. You can find **VSEScriptClient.123** in the directory where you installed the VSE Script Server. To open the file, double-click it using the Windows Explorer and you will see the **Execute Script** button as shown in Figure 178.

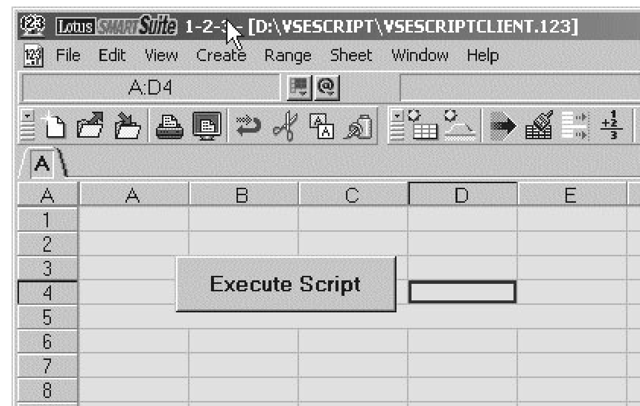


Figure 178. Sample Lotus 1-2-3 Spreadsheet for VSE Script Connector Example

Providing the VSE Script Server is running on the same workstation as your spreadsheet, you can now press **Execute Script** and the data is transferred from the VSAM cluster into the Lotus Spreadsheet as shown in Figure 179 on page 450. If the VSE Script Server is *not* running on the same workstation as your spreadsheet, you must first modify the Visual Basic script using, for example, a Script editor as shown in "How the Sample VSE Script is Defined in Lotus 1-2-3" on page 450, so that it contains the IP address or Host name of the workstation where the VSE Script Server is running.

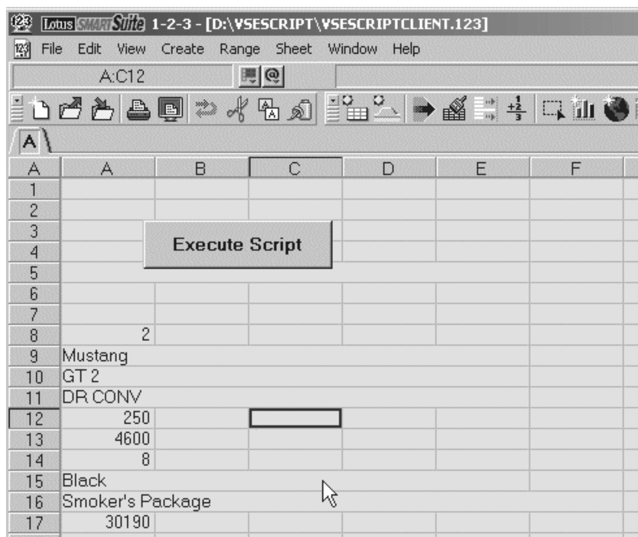


Figure 179. Transferring Data from VSAM Cluster to Lotus 1-2-3 Spreadsheet

How the Sample VSE Script is Defined in Lotus 1-2-3

Before the sample VSE Script can be used from within an office application such as Lotus 1-2-3, you must define it using, for example, a Visual Basic script (as shown in Figure 180). When you open the sample file **VSEScriptClient.123** and open the script editor, you can see the global function declarations in the “Globals” section, as shown in Figure 180.

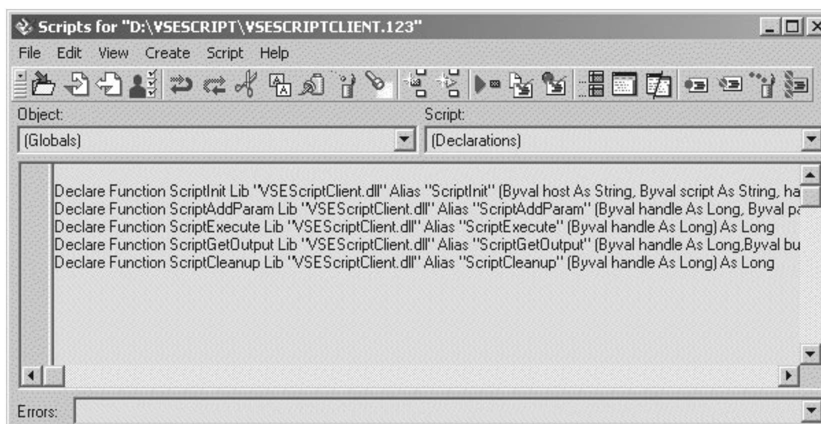


Figure 180. Sample Script As Defined in Lotus 1-2-3

You can also see the Visual Basic script in the section related to **Button 1** in Figure 181 on page 451. (In Figure 179, **Button 1** is labelled as **Execute Script**).

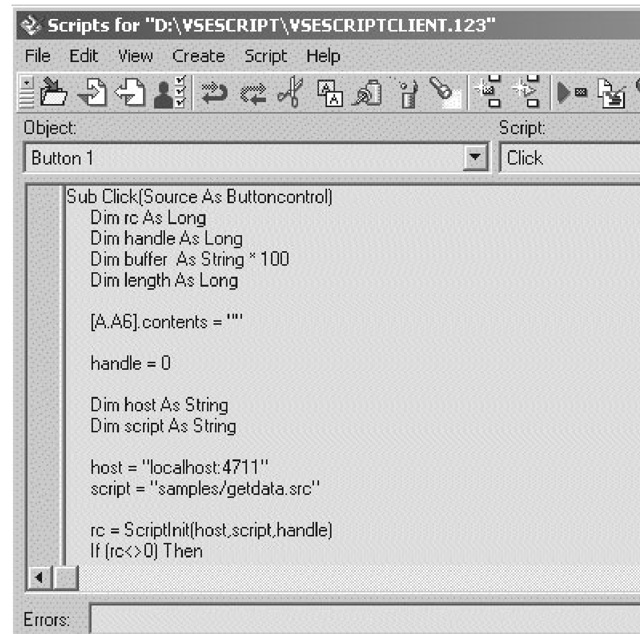


Figure 181. Visual Basic Script Used With Lotus 1-2-3 Spreadsheet Example

Note: VSEScriptClient.dll must be accessible by the office application. You may either copy the DLL into the Lotus 1-2-3 DLL directory or simply double-click the sample spreadsheet file in the same directory as the DLL.

Here is the complete Visual Basic code of the VSE Script Client, that is used to run the VSE Script in Lotus 1-2-3. The parts shown in **bold** are specific to the Lotus Spreadsheet environment.

```
Sub Click(Source As Buttoncontrol)
  Dim rc As Long
  Dim handle As Long
  Dim buffer  As String * 100
  Dim length As Long

  [A.A6].contents = ""

  handle = 0

  Dim host As String
  Dim script As String

  host = "localhost:4711"
  script = "samples/getdata.src"

  rc = ScriptInit(host,script,handle)
  If (rc0) Then
     [A.A6].contents = "rc = " &rc
     Goto finish
  End If

  rc = ScriptAddParam(handle,"2")
  If (rc0) Then
     [A.A6].contents = "rc = " &rc
     Goto finish
  End If

  rc = ScriptExecute(handle)
  If (rc0) Then
     [A.A6].contents = "rc = " &rc
```

```
      Goto finish
   End If

   Dim counter As Long
   Dim rows As Range
   Dim cell As Variant

   Set rows = Bind("A8..A65535")

   counter = 0
   Do
      rc = ScriptGetOutput(handle,buffer,100,length)
      If (rc=0) Then
         buffer = Left(buffer,length)

         Set cell = rows.Cell(counter,0,0)
         cell.contents = buffer

         counter = counter + 1
      End If
   Loop While rc=0

   Finish:

   rc = ScriptCleanup(handle)
End Sub
```

Step 7(b): Open the Sample MS Office Spreadsheet

The sample MS Office Spreadsheet file **VSEScriptClient.xls** already has the necessary setup to run the VSE Script. You can find **VSEScriptClient.xls** in the directory where you installed the VSE Script Server. To open the file, double-click it using the Windows Explorer and you will see the **Execute Script** button as shown in Figure 182.

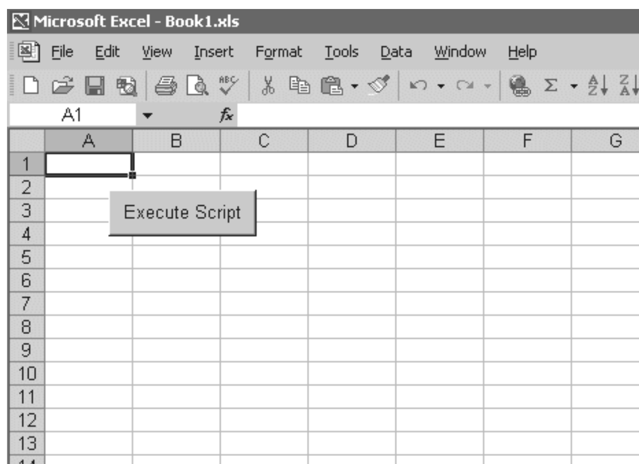


Figure 182. Sample Spreadsheet for MS Office Spreadsheet Example

Providing the VSE Script Server is running on the same workstation as your spreadsheet, you can now press **Execute Script** and the data is transferred from the VSAM cluster into the MS Office Spreadsheet as shown in Figure 183 on page 453. If the VSE Script Server is *not* running on the same workstation as your spreadsheet, you must first modify the Visual Basic script using, for example, a Script editor as shown in Figure 184 on page 453, so that it contains the IP address or Host name of the workstation where the VSE Script Server is running.

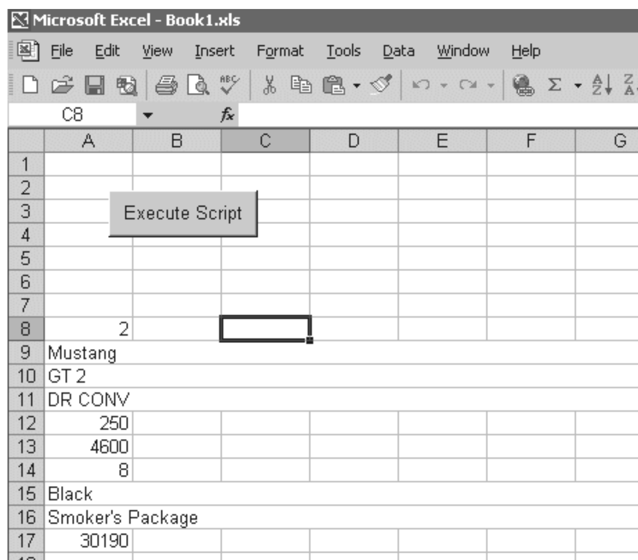


Figure 183. Transferring Data from VSAM Cluster to MS Office Spreadsheet

How the Sample VSE Script is Defined in MS Office

Before the sample VSE Script can be used from within an office application such as MS Office, you must define it using, for example, a Visual Basic script (as shown in Figure 184). When you open the sample file **VSEScriptClient.xls** and open the Visual Basic editor, you can see the global function declarations and the Visual Basic script.

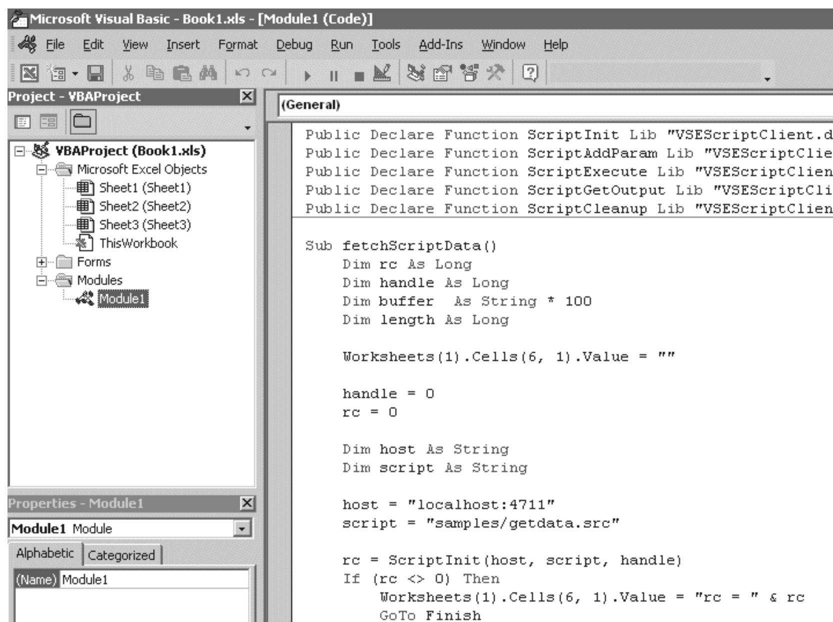


Figure 184. Sample Script as Defined in MS Office

Note: VSEScriptClient.dll must be accessible by the office application. You may either copy the DLL into the MS Office directory or simply double-click the sample spreadsheet file in the same directory as the DLL.

Here is the complete Visual Basic code of the VSE Script Client, that is used to run the VSE Script in MS Office. The parts shown in **bold** are specific to the MS Office Spreadsheet environment.

```
Attribute VB_Name = "Module1"
Public Declare Function ScriptInit Lib "VSEScriptClient.dll" (ByVal host As
   String, ByVal script As String, handle As Long) As Long
Public Declare Function ScriptAddParam Lib "VSEScriptClient.dll" (ByVal handle
   As Long, ByVal parameter As String) As Long
Public Declare Function ScriptExecute Lib "VSEScriptClient.dll" (ByVal handle
   As Long) As Long
Public Declare Function ScriptGetOutput Lib "VSEScriptClient.dll" (ByVal handle
   As Long, ByVal buffer As String, ByVal maxlen As Long, retlen As Long) As Long
Public Declare Function ScriptCleanup Lib "VSEScriptClient.dll" (ByVal handle
   As Long) As Long
Sub fetchScriptData()
    Dim rc As Long
    Dim handle As Long
    Dim buffer  As String * 100
    Dim length As Long

    Worksheets(1).Cells(6, 1).Value = ""

    handle = 0
    rc = 0

    Dim host As String
    Dim script As String

    host = "localhost:4711"
    script = "samples/getdata.src"

    rc = ScriptInit(host, script, handle)
    If (rc <> 0) Then
        Worksheets(1).Cells(6, 1).Value = "rc = " & rc
        GoTo Finish
    End If

    rc = ScriptAddParam(handle, "2")
    If (rc <> 0) Then
        Worksheets(1).Cells(6, 1).Value = "rc = " & rc
        GoTo Finish
    End If

    rc = ScriptExecute(handle)
    If (rc <> 0) Then
        Worksheets(1).Cells(6, 1).Value = "rc = " & rc
        GoTo Finish
    End If

    Dim counter As Long

    counter = 0
    Do
        rc = ScriptGetOutput(handle, buffer, 100, length)
        If (rc = 0) Then
            buffer = Left(buffer, length)

            Worksheets(1).Cells(counter + 8, 1).Value = buffer

            counter = counter + 1
        End If
    Loop While rc = 0
```

```
Finish:

    rc = ScriptCleanup(handle)
End Sub
```

Step 7(c): Start a Sample VSE Script from the Command Line

To start a sample VSE Script from the command line, you can use the batch file **runscript.bat** which is located in the directory where you installed the VSE Script Server.

For example, to start the sample script that has the name **break_cont.src**, you would enter at the command line:

```
runscript samples\break_cont.src
```

Note: The **samples** directory contains other sample scripts that you can use for learning and testing purposes.

Obtaining Data From the Middle-Tier Using VSE Script Clients

IBM provides two VSE Script Clients that run under z/VSE:
- A VSE Script Client that runs as a *batch program*.
- A VSE Script Client that runs as a *CICS client*.

You can use these VSE Script Clients to obtain data from the physical/logical middle-tier by invoking scripts inside the VSE Script Server.

From z/VSE 4.2 onwards, VSE Script Clients support *SSL-encrypted connections* for connections to the VSE Script Server. To use SSL, you must have:
- Enabled SSL in TCP/IP for VSE/ESA.
- Created the required keys and certificates on:
  – The VSE Script Server-side.
  – z/VSE.

Both VSE Script Clients:
1. Take lines with a fixed size of 80 bytes as input.
2. Return lines with a fixed size of 80 bytes as output.

- Your first input-line must be the host name of the server where the VSE Script Server is running.
- When SSL=NO, your second input-line must contain the name of the script that is to be invoked.
- When SSL=YES, your:
  – Second input-line must contain the SSL version that is to be used (for example, SSL30 or TLS31).
  – Third input-line must contain the name of the VSE Keyring Library containing the SSL members (each SSL member contains keys and certificates).
  – Fourth input-line must contain the name of the required SSL member in the VSE Keyring Library.
  – Fifth input-line must contain the SSL cipher specs to use.
  – Sixth input-line must contain the SSL session timeout, in seconds.
  – Seventh input-line must contain the name of the script that is to be invoked.
- All further input-lines consist of parameters that are supplied to the invoked script (each line is a single parameter).
- If a parameter is longer than one line, you must use *line-continuation*.

(The *SSL parameter* and *line-continuation* are explained later in this section).

These are the parameters that are used with both IBM-supplied VSE Script Clients:
- CODEPAGE
- SHOWERROR
- NOCONT
- SSL
- ALLOWCLIENTAUTH
- SYMBOLS

The **CODEPAGE** parameter specifies the code page of the z/VSE input-data and output-data:
- Since the VSE Script Server is ASCII-based, all input and output data must be translated from EBCDIC to ASCII, and vice-versa.
- The translation is done inside the VSE Script Server, using the specified code page. If a code page is not specified, the default code page IBM-1047 will be used.

The **SHOWERROR** parameter can be set to either YES or NO:
- If set to NO, the lines containing the script error (which are located at the end of the script output) will be removed.
- If set to YES (the default), the lines containing the script error (which are located at the end of the script output) will be retained.

The **NOCONT** parameter specifies whether or not line-continuation should be used for the input-lines. The NOCONT parameter can be set to either YES or NO:
- If set to NO (the default), line-continuation will be used. If you are using a parameter with a length that is longer than one line, you should set NOCONT to NO.
  - Each line that ends with a '-' will have the next line of input appended to it (the '-' will be removed).
  - If you wish to use a parameter line that ends with a '-', the line must end with two dashes (that is '--'. The next input-line must also be empty.
- If set to YES, each input-line will be used exactly as it is written.

The **SSL** parameter specifies if SSL should be used for connections to the VSE Script Server:
- When SSL=NO (the default), the connection to the VSE Script Server does *not* use SSL, and is unencrypted.
- When SSL=YES, the connection to the VSE Script Server *will be* encrypted with SSL.

The **ALLOWCLIENTAUTH** parameter specifies if SSL client authentication should be used for connections to the VSE Script Server. This parameter is only valid with when SSL=YES.
- When ALLOWCLIENTAUTH=NO (the default), the SSL connection to the VSE Script Server *does* use SSL client authentication. The only VSE Keyring Library member that is required is the .CERT.
- When ALLOWCLIENTAUTH=YES, the connection to the VSE Script Server *will allow* SSL client authentication. The VSE Keyring Library must contain .CERT, .ROOT and .PRVK members.

The VSE Script Server determines if client authentication should be performed. If the server requires client authentication but the VSE Script Client does not allow client authentication, a connection will fail.

The **SYMBOLS** parameter specifies if symbols should be resolved inside the script input text.

- When SYMBOLS=NO (the default), symbols inside the script input text are not resolved.
- When SYMBOLS=YES, any symbols in the script input text are replaced by the value defined by the symbol.

Each VSE Script file should always finish with a call to the script function `exit(<returncode>)`:

- The `returncode` provided to the `exit()` function is used as return code of the VSE job.
- If the script fails for some reason, the script error code is used as return code of the job.
- If the script does not call `exit()` and does not fail, `returncode` will be zero.

Using the VSE Script Client That Runs in Batch

The *VSE Script Client for batch* is invoked by a job. The program is IESSCBAT.phase:

1. After invocation, the VSE Script Client connects to the VSE Script Server.
2. The VSE Script Client calls the specified script, and sends all input-data lines to the script.
3. The VSE Script Client receives all output lines from the script, and writes the lines to the SYSLST of the job.
4. The return code of the invoked script is the same as the return code of the job.

Note: Some return codes that are used by the VSE Script Client indicate local errors. To ensure that it is possible to differentiate local errors from remote script errors, the invoked script should *not* use these return codes.

The VSE Script Client for batch uses the values of the three parameters contained in the job's PARM statement. If a parameter is not given in the PARM, its default is used. You specify the parameters as follows:

```
CODEPAGE=xxx
SHOWERROR=YES|NO
NOCONT=YES|NO
SSL=YES|NO
ALLOWCLIENTAUTH=YES|NO
SYMBOLS=YES|NO
```

The job's SYSIPT input contains:

- The name of the z/VSE host to which a connection is to be made (in the first line).
- If SSL=YES, *all* SSL parameters (one per line).
- The name of the script that is to be invoked (in one line).
- The parameters to be passed to the script (all remaining lines).

Here is an example of a job that invokes the VSE Script Client for batch:

```
* $$ JOB JNM=START,DISP=L,CLASS=A
// JOB START
// LIBDEF *,SEARCH=(PRD1.BASE,PRD2.SCEEBASE,PRD2.DBASE)
```

```
// EXEC IESSCBAT,PARM='CODEPAGE=Cp1047 SHOWERROR=yes'
10.31.0.1:4711
testscript.src
First parameter.
This is a longer parameter that takes two input-lines on SYSIPT. It -
is provided as a single parameter to the invoked script.
This third parameter line ends with a --
/*
/&
* $$ EOJ
```

Here is an example of a job that invokes the VSE Script Client for batch and uses SSL=YES:

```
* $$ JOB JNM=START,DISP=L,CLASS=A
// JOB START
// LIBDEF *,SEARCH=(PRD1.BASE,PRD2.SCEEBASE,PRD2.DBASE)
// EXEC IESSCBAT,PARM='CODEPAGE=Cp1047 SHOWERROR=yes SSL=yes'
10.31.0.1:4711
SSL30
CRYPTO.KEYRING
SCRPTKEY
010208090A62
86440
testscript.src
First parameter.
This is a longer parameter that takes two input lines on SYSIPT. It -
is provided as a single parameter to the invoked script.
This third parameter line ends with a --
/*
/&
* $$ EOJ
```

From z/VSE 5.1 onwards, you can use *symbolic parameters* in the SYSIPT input for the VSE Script Client being used in batch:

- These symbolic parameters will be resolved at runtime, and parameters will be replaced by the values of the symbolic parameters.
- When symbolic parameters are used in SYSIPT input, the same rules apply as when symbolic parameters are used in JCL. Therefore, for details refer to the description of symbolic parameters provided in the chapter "Job Control and Attention Routine" of the publication *z/VSE System Control Statements*.
- The parameter SYMBOLS must be set to YES to enable symbol processing. The default is NO.

Here is an example of a job that invokes the VSE Script Client for batch and uses *symbolic parameters* in the SYSIPT input:

```
* $$ JOB JNM=START,DISP=L,CLASS=A
// JOB START
// LIBDEF *,SEARCH=(PRD1.BASE,PRD2.SCEEBASE,PRD2.DBASE)
// SETPARM IP='10.31.0.1'
// SETPARM PORT='4711'
// EXEC IESSCBAT,PARM='SYMBOLS=YES CODEPAGE=Cp1047 SHOWERROR=yes'
&IP:&PORT
testscript.src
First parameter.
This is a longer parameter that takes two input lines on SYSIPT. It -
is provided as a single parameter to the invoked script.
This third parameter line ends with a --
/*
/&
* $$ EOJ
```

Listed below are the return codes that can be generated if a local error occurs in the VSE Script Client (in decimal values).

- These are the *general return codes*:
  1 ERR_NULL_PTR
  2 ERR_INVALID_HANDLE
  3 ERR_INVALID_STATUS
  4 ERR_UNKNOWN_HOST
  5 ERR_CONNECT_FAILED
  6 ERR_CONNECTION_BROKEN
  7 ERR_NO_MORE_DATA
  8 ERR_EMPTY_PARAM
  9 ERR_ICONV_OPEN_FAILED
  10 ERR_ICONV_FAILED
  11 ERR_SSL_NOT_SUPPORTED
  12 ERR_SSL_INIT_FAILED
  13 ERR_SSL_UNKNOWN_KEYNAME
  14 ERR_SSL_HANDSHAKE_FAILED
- These are the *batch client-specific return codes*:
  16 ERR_OUT_OF_MEMORY
  17 ERR_NO_SYSIPT_INPUT
  18 ERR_UNEXPECTED_END_OF_SYSIPT
  19 ERR_INVALID_PARM
  20 ERR_SYMBLIB_ERROR

Using the VSE Script Client That Runs Under CICS

The *VSE Script Client for CICS* works in a similar way to the VSE Script Client for batch. The information provided in "Using the VSE Script Client That Runs in Batch" on page 457 also applies to the VSE Script Client for CICS, *except for* how:
- The CICS client is called.
- Input/output data is provided.

The CICS client is a program named IESSCCIC. It is callable by each CICS program via an EXEC CICS LINK statement.

This is the layout of the `INPUT COMMAREA`:

```
char[12] codepage;           // codepage of input/output data
unsigned short flags;        // 0x0001=SHOWERROR
                             // 0x0002=NOCONT
                             // 0x0004=SSL
                             // 0x0008=ALLOWCLIENTAUTH
                             // 0x0010=SYMBOLS
                             // 0x1000=PRINTERRORSTOSYSLST
unsigned short int lineCount; // number of input-lines,
...input-lines...            // each 80 bytes long
```

This is the layout of the `OUTPUT COMMAREA`:

```
int clientRc;                // return code of client, see below
int scriptRc;                // return code of script
unsigned short int lineCount; // number of output lines,
...output lines...           // each 80 bytes long
```

Listed below are the return codes that can be generated if a local error occurs in the VSE Script Client (in decimal values).

- These are the *general return codes*:
  1 ERR_NULL_PTR
  2 ERR_INVALID_HANDLE

3 ERR_INVALID_STATUS
4 ERR_UNKNOWN_HOST
5 ERR_CONNECT_FAILED
6 ERR_CONNECTION_BROKEN
7 ERR_NO_MORE_DATA
8 ERR_EMPTY_PARAM
9 ERR_ICONV_OPEN_FAILED
10 ERR_ICONV_FAILED
11 ERR_SSL_NOT_SUPPORTED
12 ERR_SSL_INIT_FAILED
13 ERR_SSL_UNKNOWN_KEYNAME
14 ERR_SSL_HANDSHAKE_FAILED

- These are the *CICS client-specific return codes*:

16 ERR_OUT_OF_MEMORY
17 ERR_UNEXPECTED_END_OF_INPUT
18 ERR_INVALID_COMMAREA
19 ERR_INVALID_LINECOUNT
20 ERR_SYMBLIB_ERROR
32 WARN_OUTPUT_LINES_TRUNCATED

The field `scriptRc` is set with the return code of the invoked VSE Script file when the field `clientRc` return code is either:
- zero (no error).
- 32 (warning that the output lines were truncated).

For all other `clientRc` return codes, the field `scriptRc` is invalid.

The CICS client returns a return code in the RESP2 field after EXEC CICS LINK was called. This return code is the same as set in the field `clientRc` of the output COMMAREA, with one exception:
- If the provided COMMAREA is smaller than 8 bytes, the error codes will not fit into the COMMAREA. The COMMAREA is left unchanged and the return code ERR_INVALID_COMMAREA is returned in the RESP2 field.
- The INPUT COMMAREA contains the CODEPAGE parameter which specifies the code page of the input/output data. If the code page parameter is binary zero or empty (only blanks), the default code page is used.
- The FLAGS parameter of the INPUT COMMAREA can be used to set the SHOWERROR, NOCONT, SSL, ALLOWCLIENTAUTH, SYMBOLS and PRINTERRORSTOSYSLST parameters:
  - If the flag for a parameter is set, this means YES.
  - If the flag for a parameter is not set, this means NO.
  - For example, if you want to set SHOWERROR and NOCONT to YES, you must set the FLAGS parameter to X'0003'.
- The LINECOUNT parameter of the INPUT COMMAREA specifies the number of input-lines that are contained in the COMMAREA.
- The input-lines start after the LINECOUNT parameter.

The OUTPUT COMMAREA contains the:
- Return code of the script client.
- Return code of the invoked script.
- Number of output lines.
- The output lines start after the linecount parameter.
  - If the output lines from the invoked script do not fit into the provided COMMAREA, they will be truncated.

– If the output lines are truncated, the client return code will be set to WARN_OUTPUT_LINES_TRUNCATED.

Setting Up the VSE Script Client for CICS

- Program IESSCCIC is defined to DBDCCICS by default.
- If you need to define this program on another CICS system, you should use these options for the CEDA DEFINE PROGRAM command:

```
CEDA  DEFine PROGram( IESSCCIC )
 PROGram        : IESSCCIC
 Group          : VSESPG
 DEscription  ==>
 Language     ==> C
 RELoad       ==> Yes
 RESident     ==> No
 USAge        ==> Normal
 USEsvacopy   ==> No
 Status       ==> Enabled
 RSl            : 00
 Cedf         ==> Yes
 DAtalocation ==> Any
 EXECKey      ==> Cics
REMOTE ATTRIBUTES
 REMOTESystem ==>
 REMOTEName   ==>
 Transid      ==>
 EXECUtionset ==> Fullapi
```

Part 5. Appendixes

© Copyright IBM Corp. 2000, 2013

Glossary

This glossary defines technical terms and abbreviations used in the z/VSE e-business Connectors User's Guide. If you do not find the term you are looking for, view the *IBM Dictionary of Computing* located at: **www.ibm.com/ibm/terminology**.

The glossary includes definitions with symbol * where there is a one-to-one copy from the IBM Dictionary of Computing.

***applet**
: An application program, written in the Java programming language, that can be retrieved from a Web server and executed by a Web browser. A reference to an applet appears in the markup for a Web page, in the same way that a reference to a graphics file appears; a browser retrieves an applet in the same way that it retrieves a graphics file. For security reasons, an applet's access rights are limited in two ways: the applet cannot access the file system of the client upon which it is executing, and the applet's communication across the network is limited to the server from which it was downloaded. Contrast with *servlet*.

Asymmetric cryptography
: Synonymous with *Public key cryptography*.

***authentication**
: (1) In computer security, verification of the identity of a user or the user's eligibility to access an object. (2) In computer security, verification that a message has not been altered or corrupted. (3) In computer security, a process used to verify the user of an information system or protected resources.

bytecode
: See *Java bytecode*.

CA
: See *Certificate Authority*.

CICS ECI
: The CICS External Call Interface (ECI) is one possible requester type of the *CICS business logic interface* that is provided by the CICS Transaction Server for VSE/ESA. It is part of the CICS client and allows workstation programs to call CICS functions on the z/VSE host.

CICS EPI
: The CICS External Presentation Interface (EPI) is part of the CICS client, and enables a non-CICS Client application to act as a logical 3270 terminal and so control a CICS 3270 application.

CICS EXCI
: The EXternal CICS Interface (EXCI) is one possible requester type of the *CICS business logic interface* that is provided by the CICS Transaction Server for VSE/ESA. It allows any VSE batch application to call CICS functions.

Common Connector Framework (CCF)
: Is part of IBM's *Visual Age for Java*, and allows connections to remote hosts to be created and maintained. The CCF classes are contained in the VSEConnector.jar file and are used internally by the VSE Java Beans. CCF is important for multi-tier architectures where, for example, servlets run on a middle-tier platform. Because CCF allows open connections to be kept in a pool, this avoids the time involved in opening and closing TCP/IP connections to the remote z/VSE host each time a servlet is invoked.

***Common Object Request Broker Architecture (CORBA)**
: A specification produced by the Object Management Group (OMG) that presents standards for various types of object request brokers (such as client-resident ORBs, server-based ORBs, system-based ORBs, and library-based ORBs). Implementation of CORBA standards enables object request brokers from different software vendors to interoperate.

ConnectionManager class
: Is part of CCF, and identifies the connection to a remote z/VSE host: it holds connections between the middle-tier and the remote z/VSE server. Servlets can reserve a connection from the pool, work with it and give it back later. This is performed internally using VSE Java Beans.

© Copyright IBM Corp. 2000, 2013

connector
: In the context of z/VSE, a connector provides the middleware to connect two platforms: Web Client and z/VSE host, middle-tier and z/VSE host, or Web Client and middle-tier. These connectors are described in this publication.

container
: Is part of the JVM of application servers such as the IBM WebSphere Application Server, and facilitates the implementation of servlets, EJBs, and JSPs, by providing resource and transaction management resources. For example, an EJB developer must not code against the JVM of the application server, but instead against the interface provided by the container. The main role of a container is to act as an intermediary between EJBs and clients, and also to manage multiple EJB instances. After EJBs have been written, they must be stored in a container residing on an application server. The container then manages all threading and client-interaction with the EJBs, and co-ordinate connection- and instance pooling.

cryptographic token
: Usually referred to simply as a *token,* this is a device which provides an interface for performing cryptographic functions like generating digital signatures or encrypting data.

***cryptography**
: (1) The transformation of data to conceal its meaning. (2) In computer security, the principles, means, and methods for encrypting 'plaintext' and decrypting 'ciphertext'.

DB2-Based Connector
: Is a feature introduced with VSE/ESA 2.5, which includes a customized DB2 version, together with VSAM and DL/I functionality, to provide access to DB2, VSAM, and DL/I data, using DB2 Stored Procedures.

DB2 Stored Procedure
: In the context of z/VSE, a DB2 Stored Procedure is a Language Environment (LE) program that accesses DB2 data. However, from VSE/ESA 2.5 onwards you can also access VSAM and DL/I data using a DB2 Stored Procedure. In this way, it is possible to exchange data between VSAM and DB2.

Data Encryption Standard (DES)
: In computer security, the National Institute of Standards and Technology (NIST) Data Encryption Standard, adopted by the U.S. government as Federal Information Processing Standard (FIPS) Publication 46, which allows only hardware implementations of the data encryption algorithm.

***Decryption**
: In computer security, the process of transforming encoded text or ciphertext into plaintext.

DES See *Data Encryption Standard*.

***digital signature**
: In computer security, encrypted data, appended to or part of a message, that enables a recipient to prove the identity of the sender.

Digital Signature Algorithm (DSA)
: The Digital Signature Algorithm is the U.S. government-defined standard for digital signatures. The DSA digital signature is a pair of large numbers, computed using a set of rules (that is, the DSA) and a set of parameters such that the identity of the signatory and integrity of the data can be verified. The DSA provides the capability to generate and verify signatures.

DSA See *Digital Signature Algorithm*.

ECI See *CICS ECI*.

***Encryption**
: In computer security, the process of transforming data into an unintelligible form in such a way that the original data either cannot be obtained or can be obtained only by using a decryption process.

Enterprise Java Bean (EJB)
: An EJB is a distributed Java Bean. "Distributed" means, that one part of an EJB runs inside the JVM of a web application server, while the other part runs inside the JVM of a Web browser. An EJB either represents one data row in a database (entity bean), or a connection to a remote database (session bean). Normally, both types of an EJB work

together. This allows to represent and access data in a standardized way in heterogenous environments with relational and non-relational data. See also *Java Bean*.

EPI See *CICS EPI*.

EXCI See *CICS EXCI*.

*** FCP** See *Fibre Channel Protocol*.

***fibre-channel connection (FICON®)**
A fibre-channel communication protocol designed for IBM mainframe computers and peripherals.

***Fibre Channel Protocol (FCP)**
The serial SCSI command protocol used on fibre-channel networks.

***FICON**
See *fibre-channel connection*.

***firewall**
In communication, a functional unit that protects and controls the connection of one network to other networks. The firewall (a) prevents unwanted or unauthorized communication traffic from entering the protected network and (b) allows only selected communication traffic to leave the protected network.

hash function
A hash function is a transformation that takes a variable-size input and returns a fixed-size string, which is called the *hash value*. In cryptography, the hash functions should have some additional properties:
- The hash function should be easy to compute.
- The hash function is one-way; that is, it is impossible to calculate the 'inverse' function.
- The hash function is collision-free; that is, it is impossible that different input leads to the same hash value.

hash value
The fixed-sized string resulting after applying a *hash function* to a text.

home interface
Provides the methods to instantiate a new EJB object, introspect an EJB, and remove an EJB instantiation. As for the remote interface, only an interface is needed because the deployment tool generates the implementation class. Every Session bean's home interface must supply at least one *create()* method.

***hop** One segment of a transmission path between adjacent nodes in a routed network.

HTTP Session
In the context of z/VSE, identifies the Web-browser client that calls a servlet (in other words, identifies the connection between the client and the middle-tier platform).

***internet**
A wide area network connecting thousands of disparate networks in industry, education, government, and research. The Internet network uses TCP/IP (Transmission Control Protocol/Internet Protocol) as the standard for transmitting information.

***interface definition language (IDL)**
In CORBA, a declarative language that is used to describe object interfaces, without regard to object implementation.

JAR Is a platform-independet file format that aggregates many files into one. Multiple applets and their requisite components (.class files, images, and sounds) can be bundled in a JAR file, and then downloaded to a Web browser using a single HTTP transaction (much improving the download speed). The JAR format also supports compression, which reduces the files size (and further improves the download speed). The compression algorithm used is fully compatible with the ZIP algorithm. The owner of an applet can also digitally sign individual entries in a JAR file, to authenticate their origin.

Java applet
See *applet*.

Java application
A Java program that runs inside the JVM of your Web browser. The program's code resides on a local hard disk or on the LAN. Java applications may be large programs using graphical interfaces. Java applications have unlimited access to all your local resources.

***JavaBeans**
A platform-independent, software component technology for building reusable Java components called "beans."

Once built, these beans can be made available for use by other software engineers or can be used in Java applications. Also, using JavaBeans, software engineers can manipulate and assemble beans in a graphical drag-and-drop development environment.

Java bytecode
Bytecode is created when a file containing Java source language statements is compiled. The compiled Java code or “bytecode” is similar to any program module or file that is ready to be executed (run in a computer so that instructions are performed one at a time). However, the instructions in the bytecode are really instructions to the *Java Virtual Machine*. Instead of being interpreted one instruction at a time, bytecode is instead recompiled for each operating-system platform using a just-in-time (JIT) compiler. Usually, this enables the Java program to run faster. Bytecode is contained in binary files that have the suffix **.CLASS**.

***Java Database Connectivity (JDBC)**
An application programming interface (API) that has the same characteristics as Open Database Connectivity (ODBC) but is specifically designed for use by Java database applications. Also, for databases that do not have a JDBC driver, JDBC includes a JDBC to ODBC bridge, which is a mechanism for converting JDBC to ODBC; it presents the JDBC API to Java database applications and converts this to ODBC. JDBC was developed by Sun Microsystems, Inc. and various partners and vendors.

***Java Development Kit (JDK)**
A software package that can be used to write, compile, debug, and run Java applets and applications.

***Java Runtime Environment (JRE)**
A subset of the Java Development Kit (JDK) that contains the core executables and files that constitute the standard Java platform. The JRE includes the Java Virtual Machine, core classes, and supporting files.

***JavaScript**
A scripting language that resembles Java and was developed by Netscape for use with the Netscape browser.

Java Server Page (JSP)
A web page, similar to a HTML page. Parts of a JSP are compiled into a servlet by the web server's JSP engine while sending the web page to the requesting browser. JSPs have the advantage, that the developer does not have to recompile the servlet each time it is changed. Changes are always made in the JSP file. A JSP also creates web pages dynamically.

Java servlet
See *servlet*.

***Java Virtual Machine (JVM)**
A software implementation of a central processing unit (CPU) that runs compiled Java code (applets and applications).

Lotus Domino Server
The Lotus Notes® server which is used for storing each user's *Composite Database*.

Multipurpose Internet Mail Extensions (MIME)
An Internet standard for identifying the type of object that is transferred across the Internet. MIME types include several variants of audio, graphics, and video.

***nonce**
(1) A random, unique text string that is encrypted along with data and then is used to detect attacks against the system that sends the encrypted data. A nonce is used especially for authentication and ensures that encrypted data is different each time that it is encrypted. (2) A unique cryptographic number that is embedded in a message to help detect a replay attack.

persistence
A term used to describe the storage of objects in a database to allow them to persist over time, rather than being destroyed when the application containing them terminates. Enterprise Java Bean containers, such as WebSphere, provide persistence services for EJBs deployed within them.

persistent storage
See *Persistence*.

PKCS (Public Key Cryptography Standards)
PKCS is a set of standards, issued by RSA

Data Security, Inc., for implementation of public key cryptography.

PKI See *Public key infrastructure*.

***port** A connector on a device to which cables for other devices such as display stations and printers are attached.

***private key**
In computer security, a key that is known only to the owner. See *Public key cryptography*.

***proxy server**
A server that receives requests intended for another server and that acts on the client's behalf (as the client's proxy) to obtain the requested service. A proxy server is often used when the client and the server are incompatible for direct connection (for example, when the client is unable to meet the security authentication requirements of the server but should be permitted some services).

***public key**
In computer security, a key made available to anyone who wants to encrypt information. See *Public key cryptography*.

***public key cryptography**
In computer security, cryptography in which a public key is used for encryption and a private key is used for decryption. Synonymous with *Asymmetric cryptography*.

remote method invocation (RMI)
Is a Java-only version of CORBA. Because RMI is specific to Java, it is easier to use than CORBA. Instead of writing IDL to describe objects, a program called **rmic** can be run on Java class files, which then creates stub and skeleton classes directly from the class files. RMI is used, for example, by EJBs to communicate between the client stub and the server part of the EJB.

remote interface
In the context of z/VSE, the remote interface allows a client to make method calls to an EJB although the EJB is on a remote z/VSE host. The container uses the remote interface to create client-side stubs and server-side proxy objects to handle incoming method calls from a client to an EJB.

***remote procedure call (RPC)**
(1) A facility that a client uses to request the execution of a procedure call from a server. This facility includes a library of procedures and an external data representation. (2) A client request to a service provider located in another node.

RSA algorithm
Public key algorithm named after its inventors Rivest, Shamir, and Adleman.

secret key
Synonymous with *private key*.

Secure Electronic Transaction (SET)
An open specification for securing payment card transactions over open networks such as the Internet. SET was developed by Visa and MasterCard with participation from several technology companies, such as GTE, IBM, Microsoft, Netscape, SAIC, Terisa Systems, and VeriSign. SET will be based on encryption technology from RSA Data Security.

Secure Sockets Layer (SSL)
A security protocol that allows the client to authenticate the server and all data and requests to be encrypted. SSL was developed by Netscape Communications Corp. and RSA Data Security, Inc..

***servlet**
An application program, written in the Java programming language, that is executed on a Web server. A reference to a servlet appears in the markup for a Web page, in the same way that a reference to a graphics file appears. The Web server executes the servlet and sends the results of the execution (if there are any) to the Web browser. Contrast with *applet*.

SET See *Secure Electronic Transaction*.

skeleton
In the context of CORBA, is the equivalent of a stub but used on the server (z/VSE host). It reassembles the data that arrives from the network into a recognizable format, calls a server-side method, and sends a reply back to the client.

***socks enabled**
Pertaining to TCP/IP software, or to a specific TCP/IP application, that understands the *socks protocol*. "Socksified" is a slang term for socks-enabled.

***socksified**
: See *socks enabled*.

***socks protocol**
: A protocol that enables an application in a secure network to communicate through a firewall via a *socks server*.

***socks server**
: A circuit-level gateway that provides a secure one-way connection through a firewall to server applications in a nonsecure network.

SSL See *Secure Sockets Layer*.

***thread**
: A stream of computer instructions that is in control of a process. A multithreaded process begins with one stream of instructions (one thread) and may later create other instruction streams to perform tasks.

Uniform Resource Locator (URL)
: (1) A sequence of characters that represent information resources on a computer or in a network such as the Internet. This sequence of characters includes (a) the abbreviated name of the protocol used to access the information resource and (b) the information used by the protocol to locate the information resource. For example, in the context of the Internet, these are abbreviated names of some protocols used to access various information resources: http, ftp, gopher, telnet, and news; and this is the URL for the IBM home page: `http://www.ibm.com`. (2) The address of an item on the World Wide Web. It includes the protocol followed by the fully qualified domain name (sometimes called the host name) and the request. The Web server typically maps the request portion of the URL to a path and file name. For example, if the URL is `http://www.networking.ibm.com/nsg/nsgmain.htm`, the protocol is `http`, the fully qualified domain name is `www.networking.ibm.com`, and the request is `/nsg/nsgmain.htm`.

URL See *Uniform Resource Locator*.

VSE Connector Server
: Is the host part of the VSE Java Beans, and is started using the job STARTVCS which is placed in the VSE/POWER reader queue during installation of z/VSE. Runs by default in dynamic class R.

VSE Java Beans
: Are Java Beans that allow access to all VSE-based file systems (VSE/VSAM, Librarian, VSE/POWER, and VSE/ICCF), submit jobs, and access the z/VSE operator console. The class library is contained in the *VSEConnector.jar* archive. See also *JavaBeans*.

WebSphere Application Server
: Is used together with an IBM HTTP Server to process servlets, JSPs, and EJBs. Also requires a specific JDK and DB2.

Index

Numerics

2-tier environments
  description 7
  example of use 7
  overview diagram 7
  using applets 187
  using CICS connectivity 13
  using Java-based connector 13
  using VSAM Redirector connector 13
3-tier environments
  Database Call Level Interface 14
  description 8
  example of use 8
  overview diagram 8
  programs installed on physical/logical middle-tier 20
  storing session information 230
  using applets 189
  using CICS connectivity 14
  using DB2-based connector 14
  using EJBs 254
  using Java-based connector 14
  using JSPs 245
  using servlets 227
  VSAM Redirector connector 14
  WebSphere MQ connectivity 14

A

accessibility xvii
accessing
  DL/I data using AIBTDLI interface 403
  DL/I data using DB2 Stored Procedure 402
  DL/I data using VSE Java Beans 159
  ICCF data using VSE Java Beans 168
  Librarian using VSE Java Beans 165
  operator console using VSE Java Beans 155
  performance data via VSE Health Checker 175
  POWER data using VSE Java Beans 163
  VSAM data using DB2 Stored Procedure 395
  VSAM data using VSE Java Beans 156
  VSE data using applets 187
  VSE data using JSPs 245
  VSE data using servlets 227
  VSE data using VSE Java Beans 142
  VSE data via VSE Navigator 170
action codes for VSE Connector Server
  PLGACT_CHECKCANCEL 282
  PLGACT_FINISH 282
  PLGACT_NOTHING 282
  PLGACT_RECEIVE 282
  PLGACT_SEND 282
  PLGACT_SEND_RESP 282
action codes for VSE Connector Server *(continued)*
  PLGACT_WAIT 282
  PLGACT_WAITRECV 282
  PLGACT_WAITTIMER 282
  supported by VSE Connector Server 282
AIBTDLI
  creating programs that use 405
  format of database and checkpoint call 407
  format of Roll Back call 407
  format of scheduling call 406
  format of termination call 407
  invoking 406
AIBTDLI callable interface 403
AIBTDLI Interface
  and return code X'FF00' 409
  callable interface AIBTDLI 403
  compiling & link-editing programs 408
  error messages description 411
  errors with no return code 409
  for accessing DL/I data 403
  format of AIB 408
  partition layout 403
  return & status codes 408
  single / multiple MPS systems 409
  task termination & abend handling 410
Apache Server 20
APAR PQ39683 37
API environment
  DBCLI 291
API Environment, DBCLI 292
applet sample, data-mapping 192
applet sample,VSAM-DB2 203
applets
  disadvantages & restrictions of 191
  for accessing DL/I data via a DB2 Stored Procedure 214
  for accessing VSAM data via a DB2 Stored Procedure 201
  for defining a map 192
  in 2-tier environments 187
  in 3-tier environments 189
  sample applet 187
  using VSEAppletServer 191
  VSAM data mapping example, description 192
  VsamSpaceUsage 192
AppletViewer 200
Application Framework for e-business
  and core applications 3
  and Secure Electronic Transaction (SET) 3
  and Secure Sockets Layer (SSL) 3
  extending/levering core applications 3
  purpose 3
Application Interface Block (DLIAIB) 408
application server 20
archive tag 191
ASCII, considerations in plugins 289
Assembler programs
  using with DBCLI 298

B

BINDCOLUMN
  DBCLI function 301
binding columns
  DBCLI programming 391
BINDPARAMETER
  DBCLI function 304

C

C programs
  using with DBCLI 297
callback mechanism, of VSE Java Beans 146
CEDA transaction, for defining SOAP server 435
CEDA transaction, for defining SOAP service 434
CEDA, using for SOAP sample 434
CICS connectivity
  what it offers 5
CICS Transaction Gateway
  using for CICS connectivity in 3-tier env. 5
CICS Transaction Server
  and ECIRequest class 5
  and EPIRequest class 5
  and the CICS Universal Client 5
  and the JavaGateway 5
  customizing 84
  the VSAM-via-CICS service 99
  using for CICS connectivity in 3-tier env. 5
CICS Universal Client 5
  using for CICS connectivity in 3-tier env. 5
CleanupHandler function 281
CleanupPlugin function 276
CLI (Call Level Interface) 394, 397
CLI interface (for C programs)
  activities on the requestor 397
  activities on the z/VSE host 398
  example of syntax (VSAMSQLCloseTable) 399
  program flow when using 399
  supported functions for accessing mapped VSAM data 398
  supported SQL statements 400
Client Configuration Assistant (CCA) 96
CLOSECURSOR
  DBCLI function 307

© Copyright IBM Corp. 2000, 2013

CLOSESTATEMENT
  DBCLI function 308
COBOL programs
  using with DBCLI 296
codebase tag 191
COMMIT
  DBCLI function 308
configuration PHASE for VSAM Redirector Connector 61
configuring
  DBCLI Server 76
CONNECT
  DBCLI function 309
connection pool (DBCLI)
  querying 81
connection pool manager (DBCLI)
  configuring 78
  starting 79
  stopping 80
connection pooling
  DBCLI 78
connection pooling (DBCLI)
  overview 74
  programming concept 295
connection possibilities under z/VSE
  2- and 3-tier environments 7
  choosing connectors in 3-tier env. 14
  downloading client-part 25, 75
  downloading host-part 63
  installing DBCLI server 75
  installing VSE Connector Client 24
  overview of what connectors can be used for 13
  products it supports 3
  uninstalling DBCLI server 76
  uninstalling VSE Connector Client 27
connections properties file, example for VSE Script Server 446
Connections.properties 43, 446
CONNECTSSL
  DBCLI function 311
console functions for VSE Script Language 443
container (EJB) 249, 250
copybook, for Trap COBOL and PL/I Batch interface 126
CORBA 252
CSVFileHandler 67
cursors
  with DBCLI 294

D

data-mapping applet (sample)
  activities required on Host 194
  calling 195
  creating file index.html 194
  deploying 194
  description 192
  for adding map to VSAM cluster 196
  for modifying a map 197
  for modifying data fields of map 198
  initializing 196
  running using AppletViewer 200
  setting up the class 195
Database Call Level Interface, see DBCLI 291
DATAEX 288
date-format, for VSE Connector Server 33
DB2 Connect
  used for accessing VSAM data 395
DB2 Connect, install / configure 96
DB2 Server for VSE 394
DB2 Stored Procedures
  accessing DL/I data via 402
  accessing VSAM data via 395
  advantages 393
  for accessing VSAM data 395
  how you use 393
  interfaces you can use 394
  programming requirements for 394
DB2-based connector
  customizing 85
  define sample database 85
  description 83, 393
  using in 3-tier env. 14
DB2ConnectorJDBCApplet.java 208
DB2CTVAR job, catalog ARISIVAR.Z 86
DB2DEFCT job, define user catalog 85
DB2DLIConnectorJDBCApplet.java 220
DB2DRDA job, activate DRDA support 87
DB2DRDA job, set up for DRDA support 93
DB2Handler 67
DB2Handler, sample VSAM Redirector Handler 70
DB2JMGR, job manager 87
DB2SPSCA job, set up Stored Procedure Server, define to DB2 93
db2vsewm database-alias 201
DBATTRIBUTES
  DBCLI function 313
DBBESTROWIDENT
  DBCLI function 315
DBCATALOGS
  DBCLI function 317
DBCLI
  API environment 291
  configuring for connection pooling 78
    defining programs to CICS 79
  connection pooling with DBCLI
    defining programs to CICS 79
  cursors 294
  DBCLI server
    connecting to, disconnecting from 292
  execute an SQL statement 293
  finding cause of errors 391
  logical unit of work 292
  meta data 295
  overview 73
  performance 390
  prerequisites for 74
  programming concepts 291
  programming interface restrictions 296
  return codes 392
  using within a C program 297
  using within a COBOL program 296
  using within a PL/I program 297
  using within a REXX program 299
DBCLI *(continued)*
  using within an Assembler program 298
DBCLI function call
  syntax and parameters 299
DBCLI functions
  BINDCOLUMN 301
  BINDPARAMETER 304
  CLOSECURSOR 307
  CLOSESTATEMENT 308
  COMMIT 308
  CONNECT 309
  CONNECTSSL 311
  DBATTRIBUTES 313
  DBBESTROWIDENT 315
  DBCATALOGS 317
  DBCOLUMNPRIV 318
  DBCOLUMNS 320
  DBCROSSREFERENCE 322
  DBEXPORTEDKEYS 325
  DBIMPORTEDKEYS 327
  DBINDEXINFO 329
  DBPRIMARYKEYS 332
  DBPROCEDURECOLS 333
  DBPROCEDURES 335
  DBSCHEMAS 337
  DBSUPERTABLES 338
  DBSUPERTYPES 339
  DBTABLEPRIV 341
  DBTABLES 343
  DBTABLETYPES 345
  DBTYPEINFO 346
  DBUDTS 347
  DBVERSIONCOLS 349
  DISCONNECT 350
  EXECUTE 351
  FETCH 352
  GETCOLUMNINFO 353
  GETCONNATTR 355
  GETENVATTR 369
  GETLASTERROR 371
  GETMORERESULTS 372
  GETNUMCOLUMNS 373
  GETNUMPARAMETERS 374
  GETPARAMETERINFO 374
  GETROWNUMBER 376
  GETSTMTATTR 376
  GETUPDATECOUNT 378
  INITENV 379
  INITSSL 380
  PREPARECALL 381
  PREPARESTATEMENT 382
  RELEASESAVEPOINT 384
  roadmap of where used 300
  ROLLBACK 385
  SETCONNATTR 385
  SETENVATTR 386
  SETPOS 387
  SETSAVEPOINT 388
  SETSTMTATTR 389
  TERMENV 389
DBCLI programming
  binding columns 391
  pre-fetching 390
  SSL 390
DBCLI Server
  commands for using 81

DBCLI Server *(continued)*
configuring 76
installing 75
uninstalling 76
DBCLI Server (DBCliServer)
connecting to, disconnecting from 292
DBCOLUMNPRIV
DBCLI function 318
DBCOLUMNS
DBCLI function 320
DBCROSSREFERENCE
DBCLI function 322
DBEXPORTEDKEYS
DBCLI function 325
DBHandler 67
DBIMPORTEDKEYS
DBCLI function 327
DBINDEXINFO
DBCLI function 329
DBPRIMARYKEYS
DBCLI function 332
DBPROCEDURECOLS
DBCLI function 333
DBPROCEDURES
DBCLI function 335
DBSCHEMAS
DBCLI function 337
DBSUPERTABLES
DBCLI function 338
DBSUPERTYPES
DBCLI function 339
DBTABLEPRIV
DBCLI function 341
DBTABLES
DBCLI function 343
DBTABLETYPES
DBCLI function 345
DBTYPEINFO
DBCLI function 346
DBUDTS
DBCLI function 347
DBVERSIONCOLS
DBCLI function 349
DeltaLoader 69
disability xvii
DISCONNECT
DBCLI function 350
Distributed Relational Database Architecture
used with DB2-based connector 83
DL/I applet (sample)
and DLIREAD (sample DB2 Stored Procedure) 222
calling 218
client-side program 220
compiling DB2 Stored Procedure for 217
creating JAR file for 217
defining DB2 Stored Procedure for DB2 Server for VSE 217
defining DL/I database for 217
HTML file for calling 216
main window for 218
prerequisite steps for 216
DL/I data
accessing using AIBTDLI interface 403
accessing using DB2 Stored Procedures 402
accessing using VSE Java Beans 159
configuring to access DL/I data via VSE Java Beans 37
DL/I database, for the sample DL/I applet 217
DLIREAD, compiling 217
DLIREAD, how used 214
DLZBSEOT (task termination exit) 37, 84
DLZLX000 405
DLZMPX00 (AIBTDLI interface) 403
DR2JMGR job, install DB2 sample database 89
DR2JMGR job, prepare DB2 sample DB 88
DRDA application 393

E

e-business applications
description 4
EBCDIC, considerations in plugins 289
ECI Interface 5
ECIRequest 5
Enterprise Java Bean (EJB)
accessing from an EJB client 265
and CORBA 252
and EJB container 249
architecture, overview 249
as entity beans 249
as session beans 249
compared to JavaBeans and servlets 251
deploying 264
description 249
entities involved in EJB method call 252
example of implementing 256
for implementing client applications 252
representing VSE data using 249
sample, compile Java source files 264
sample, create record layout for employees 257
sample, defining VSAM cluster for 256
sample, implement EJB code 259
sample, implement RecordPK class 259
sample, specify EJB home interface 258
sample, specify EJB remote interface 258
used in 3-tier environment 254
entity bean 142
entity beans, properties 249
EPI Interface 5
EPIRequest 5
errors
DBCLI 391
EXECUTE
DBCLI function 351
ExecuteHandler function 279

F

FETCH
DBCLI function 352
functions used with plugins 269

G

GDPS (Geographically Dispersed Parallel Sysplex) 129
GDPS Client 129
GDPS support
commands available 132
configuring 130
configuring via SKGDPSCF job 130
diagram 129
overview 129
starting 131
starting via SKSTGDPS job 131
general functions for VSE Script Language 441
GETCOLUMNINFO
DBCLI function 353
GETCONNATTR
DBCLI function 355
getdata.src (sample VSE Script) 447
GETENVATTR
DBCLI function 369
GetHandledCommands function 277
GETLASTERROR
DBCLI function 371
GETMORERESULTS
DBCLI function 372
GETNUMCOLUMNS
DBCLI function 373
GETNUMPARAMETERS
DBCLI function 374
GETPARAMETERINFO
DBCLI function 374
getquote.c 433
getquote.c (SOAP service) 428
GetQuote.java 433
GetQuote.java (SOAP Java client) 432
GETROWNUMBER
DBCLI function 376
GETSTMTATTR
DBCLI function 376
GETUPDATECOUNT
DBCLI function 378
grouping 394

H

home interface, of EJB 258
homepage
DB2 96
for downloading DBCLI server 75
for downloading Java code 20
for downloading VSAM Redirector Server 63
for downloading VSE Connector Client 25
for downloading VSE Script Server 40

homepage *(continued)*
for WebSphere Application Server 6
VSE xix
homepage, VSE xix
HTML file for calling the DL/I applet 216
HTML file for calling the VSAM applet 203
HTTPServletRequest 230
httpSession 230

I

IBM HTTP Server, installing 20
ICCF data
accessing using VSE Java Beans 168
iesincon.w 24, 75
IESMAPD 104
IESMTRAP phase, for SNMP Trap Client in a batch job 121
IESPLGIN.H 285
IESPLGSK.C 285
iesscrpt.w, obtaining 40
IESSOAPD (SOAP decoder) 418
IESSOAPE (SOAP encoder) 420
IESSOAPH.H header file for SOAP 421
init() method, example 209
INITENV
DBCLI function 379
INITSSL
DBCLI function 380
installing
DBCLI Server 75
installing DBCLI server 75
installing VSE Connector Client 24
installing/ configuring your system
choosing connectors in 3-tier env. 14
configuring DL/I for access via VSE Java Beans 37
configuring TCP/IP 19
configuring the VSAM-via-CICS service 99
configuring the VSE Connector Server 27
connectivity possibilities 13
for 2 and 3-tier environments 7
installing DBCLI server 75
installing Java 19
installing prerequisite programs 19
installing the VSE Script Server 39
installing VSAM Redirector Server 63
installing VSE Connector Client 24
installing VSE Health Checker 177
installing VSE Navigator 173
installing WebSphere Application Server 21
programs installed on physical/logical middle-tier 20
the VSE HTTP Server 19
using the Java-based connector (2-tier env.) 13
VSAM Redirector Client / VSAM Capture Exit 49
Internet address
for WebSphere Application Server 6
VSE homepage xix

J

J2EE package, installing 435
JAR file, for DL/I applet 217
JAR file, for VSAM applet 206
Java
choosing between JDK and JRE 20
downloading the Java base code 20
downloading the Java Development Kit 20
installing & configuring 19
installing JDK 19
installing Swing classes 19
SOAP client, compile 435
SOAP client, description 432
SOAP client, running 436
SOAP-client packages, downloading 433
Java Database Connectivity 179
Java Development Kit (JDK)
installing 19
Java Runtime Environment (JRE) 20
Java Server Page (JSP)
accessing VSE data using 245
advantages in using 245
example of 247
Java-based connector
description 23
extending, with plugins 269
overview of client-part 23
overview of server-part 24
protocol used by 288
using in 2-tier env. 13
using in 3-tier env. 14
what it is 4
JavaBeans
compared to EJBs 251
Javadoc, example for VSE Java Bean 145
JavaGateway 5
JDBC (Java database connectivity) 397
JDBC (Java Database Connectivity)
advantages of 179
example of use 182
specifying table names 182
SQL statements supported 179
JDBC classes, importing 208
JDBC driver class, loading 209
JMibBrowser 121
jobs supplied
DB2CTVAR (catalog ARISIVAR.Z) 86
DB2DEFCT (define user catalog) 85
DB2DRDA (activate DRDA support) 87
DB2DRDA (set up for DRDA support) 93
DB2JMGR (job manager) 87
DB2SPSCA (set up Stored Procedure Server, define to DB2) 93
DR2JMGR (install DB2 sample database) 89
DR2JMGR (prepare DB2 sample DB) 88
IESMASCF, (replace VSE Monitoring Agent configuration file) 117
PSERVER 87
SKGDPSCF, (configuring GDPS Client) 130
jobs supplied *(continued)*
SKSTGDPS, (starting GDPS Client) 131
SKSTMAS, (starting VSE Monitoring Agent) 117

L

Librarian
accessing using VSE Java Beans 165
Librarian functions for VSE Script Language 443
logical unit of work, DBCLI
COMMIT / ROLLBACK functions 292
Lotus 1-2-3 spreadsheet file, sample 449
Lotus 1-2-3, example using a VSE Script Client 445
Lotus Domino 3
Lotus Domino Go Server 20

M

map, creating 103
mapping VSAM data
creating a map 103
creating a view 103
why you need to do so 103
maps
creating using the VSAM MapTool 112
defining 104
defining using a Java application 106
defining using RECMAP 105
defining using sample applet 105
example of creating a local VSAM map object 106
example of creating a view for a map 106
example of creating data fields for a map 106
example of displaying properties of a map 106
example of how to delete 106
storing on z/VSE host 104
structure of 104
MessageDialog.java 208
meta data
with DBCLI 295
middle-tier
programs you install 20
monitoring plug-ins, creating your own 127
MPS systems 409
MQClient mode 57
MQLoader 69
MQServer mode 57
MS Office spreadsheet, sample 452
MS Office, example using a VSE Script Client 445

N

Navigator, VSE 170
non-Java access using VSE Script Connector 437

O

ODBC (Open DataBase Connectivity) 394, 397
OIDs (Object Identifiers), used by monitoring plug-ins 119
online documentation, description 26
operator console (VSE)
  accessing using VSE Java Beans 155
overview
  connection pooling 74
  DBCLI 73

P

PL/I programs
  using with DBCLI 297
PLGACT_CHECKCANCEL action code 282
PLGACT_FINISH action code 282
PLGACT_NOTHING action code 282
PLGACT_RECEIVE action code 282
PLGACT_SEND action code 282
PLGACT_SEND_RESP action code 282
PLGACT_WAIT action code 282
PLGACT_WAITRECV action code 282
PLGACT_WAITTIMER action code 282
plugin functions
  CleanupHandler 281
  CleanupPlugin 276
  ExecuteHandler 279
  GetHandledCommands 277
  PluginMainEntryPoint 274
  SetupHandler 278
  SetupPlugin 275
PluginMainEntryPoint function 274
plugins
  and protocol for transferring data 290
  ASCII / EBCDIC considerations 289
  Big / Little Endian considerations 289
  choosing access method for 289
  compiling 285
  design considerations 288
  implementing server plugin 269
  structure of 269
  structuring client-part view 290
POWER functions for VSE Script Language 443
PowerGridLayout class 196
PowerGridLayout.java 208
pre-fetching
  DBCLI programming 390
PREPARECALL
  DBCLI function 381
PREPARESTATEMENT
  DBCLI function 382
prerequisites
  for the DBCLI 74
programming concept
  connection pooling 295
properties file for VSAM Redirector Server 65
properties file, example for VSE Script Server 446
protocol
  extending with own commands 288
  used by Java-based connector 288
protocol, between VSE Script Client / VSE Script Server 439
protocol, for communicating between client and server 290
PSERVER job 87

R

RECMAP command 105
RecordPK class 259
redbooks, to which can refer xix
redirector handler 65
Redirector, VSAM 45
RedirLoader 69
reference information
  DBCLI functions 300
RELEASESAVEPOINT
  DBCLI function 384
remote interface, of EJB 258
rename() method 197
restrictions
  with DBCLI 296
return code X'FF00' 409
return codes
  DBCLI 392
return codes, for Trap Client API 127
REXX programs
  using with DBCLI 299
ROLLBACK
  DBCLI function 385

S

Sample script, running from command line 455
sample VSE Script (getdata.src) 447
samples provided with z/VSE e-business connectors
  applet 187
  applet for VSAM data mapping 192
  DL/I applet 214
  EJB 256
  servlet 230
  VSAM applet 201
Script Server 437
Secure Electronic Transaction (SET) 3
  glossary description 469
  supported by Application Framework for e-business 3
Secure Sockets Layer (SSL)
  example for connecting to z/VSE host 149
  glossary description 469
  supported by Application Framework for e-business 3
security and the VSE Monitoring Agent 118
server
  VSEAppletServer 191
servlet
  accessing VSE data using 227
  compared to EJBs 251
  compiling and calling 229
servlet *(continued)*
  in 3-tier environments 227, 230
  sample servlet, create new flight 241
  sample servlet, create new order 242
  sample servlet, creating VSAM clusters for 231
  sample servlet, description 230
  sample servlet, display flight properties 236
  sample servlet, displaying list of flights 234
  sample servlet, get flight instances 235
  sample servlet, HTML constructs used 232
  sample servlet, place an order 239
  sample servlet, using forms to get input 232
session bean 142
session beans, properties 249
session information, storing 230
SETCONNATTR
  DBCLI function 385
SETENVATTR
  DBCLI function 386
SETPOS
  DBCLI function 387
SETSAVEPOINT
  DBCLI function 388
SETSTMTATTR
  DBCLI function 389
SetupHandler function 278
SetupPlugin function 275
SKCPSTP, compile DB2 Stored Procedures 95
SKCRESTP, create DB2 Stored Procedures 95
SKDB2SPS, catalog startup job SPSERV01 93
SKDB2STR, put DB2START in POWER reader 96
SKDB2VAR, customize DB2-based connector 85
SKDLICMP, compile/link COBOL stored procedures 95
SKDLISMP (initialize DL/I sample database) 37, 95
SKDLISTP, create stored procedures for DL/I access 95
skeletons
  SKCPSTP (compile DB2 Stored Procedures) 95
  SKCRESTP (create DB2 Stored Procedures) 95
  SKDB2SPS (catalog startup job SPSERV01) 93
  SKDB2STR (put DB2START in POWER reader) 96
  SKDB2VAR (customize DB2-based connector) 85
  SKDLICMP (compile/link COBOL stored procedures) 95
  SKDLISMP, initialize DL/I sample database 37, 95
  SKDLISTP (create stored procedures for DL/I access) 95

skeletons *(continued)*
SKVSSAMP (define VSE/VSAM cluster, load data) 95
SKGDPSCF, job for configuring GDPS Client 130
SKJOURN, and CICS TS 84
SKSTGDPS, job for starting GDPS Client 131
SKSTMAS, job for starting VSE Monitoring Agent 117
SKVCSCAT, VSE Connector Server catalog members job 28
SKVCSCFG, specify general settings for VSE Connector Server 29
SKVCSLIB, specify libraries for VSE Connector Server 31
SKVCSPLG, specify plugins for VSE Connector Server 31
SKVCSSSL, configure VSE Connector Server for SSL 32
SKVCSSTJ, for placing startup job in reader queue 28
SKVCSUSR, and VSE Connector Server 36
SKVCSUSR, specify logon access for VSE Connector Server 32
SKVSSAMP job 204
SKVSSAMP sample skeleton 446
SKVSSAMP, define VSE/VSAM cluster, load data 95
SNMP clients 117
SNMP command-line tools 121
SNMP requests 115
SNMP Trap Client API
using in a batch job 121
using in CICS programs 125
using in COBOL and PL/I programs 124
using in LE/C programs 123
SNMP Trap Client, symbolic parameters 121
SOAP (Simple Object Access Protocol)
and COMMAREA 418, 420
client (IBM-supplied), code for 430
compile /link sample C programs 434
control blocks 421
converter 418, 420
decoder (IESSOAPD) 418
define SOAP server to CICS (CEDA) 435
description 414
encoder (IESSOAPE) 420
general syntax 414
header file IESSOAPH 421
Java client (IBM-modified), code for 432
Java SOAP-client packages 433
running IBM-supplied sample 433
service (IBM-supplied), code for 428
writing own programs 436
z/VSE host acting as SOAP client 420
z/VSE host acting as SOAP server 418
SOAP client 420
SOAP client (C-program), running 436
SOAP client (Java), compile 435
SOAP client (Java), running 436
SOAP server 418
SOAP_DEC_PARAM control block 425
SOAP_PARAM_HDR control block 422
SOAP_PROG_PARAM control block 424
soapclnt.c 433
soapclnt.c (SOAP client) 430
SQL
comparison with VSE Java Beans terminology 182
statements supported by JDBC 179
statements supported by VSAMSQL CLI 400
SQL statements
executing with DBCLI 293
sqlds database 201
SSL
DBCLI programming 390
SSL parameter 455
STARTVCS job 34
Stored Procedure Server
description 394
string functions for VSE Script Language 442
symbolic parameters, in a batch VSE Script Client program 457
symbolic parameters, used with SNMP Trap Client 121
syntax and parameters
DBCLI function call 299
system activity, displaying using VSE Navigator 170

T

TCP/IP
configuring on the z/VSE host 19
customizing 85
TCP/IP connections, and VSE Monitoring Agent 117
TERMENV
DBCLI function 389
Trap Client API (SNMP), in CICS programs 125
Trap Client API (SNMP), in COBOL and PL/I programs 124
Trap Client API (SNMP), in LE/C programs 123
Trap Client API, return codes 127
Trap COBOL and PL/I Batch interface, copybook for 126
Trap LE/C interface, functions provided by 124

U

uninstalling
DBCLI Server 76
uninstalling DBCLI server 76
uninstalling VSE Connector Client 27
UTF-8 439

V

view, creating 103
views
example of adding data fields 106
example of creating a view for a map 106
example of displaying properties 106
VisualAge for Java 3
VSAM applet (sample)
calling 206
client-side program 208
compiling DB2 Stored Procedure for 203
creating JAR file for 206
defining DB2 Stored Procedure for DB2 Server for VSE 204
defining VSAM data cluster for 204
general description 201, 214
HTML file for calling the applet 203
main window for 206
prerequisite steps for 203
server-side DB2 Stored Procedure 210
VSAM Capture Exit
configuring 49
creating delta records 56
VSAM Capture Exit, redirection modes 55
VSAM data
accessing using DB2 Stored Procedure 395
accessing using JDBC 179
accessing using VSE Java Beans 156
accessing via DB2 Stored Procedures 395
creating maps using the VSAM MapTool 112
defining maps 104
defining maps using a Java application 106
defining maps using RECMAP 105
defining maps using sample applet 105
example of adding data fields to a view 106
example of creating a local VSAM map object 106
example of creating a view for a map 106
example of creating data fields for a map 106
example of displaying properties of a map 106
example of displaying properties of a view 106
program flow when using CLI 399
storing maps on z/VSE host 104
structure of VSAM maps 104
supported CLI functions for accessing mapped data 398
VSAM data cluster, for the sample VSAM applet 204
VSAM Delta Cluster, using 56
VSAM functions for VSE Script Language 444
VSAM MapTool 112
VSAM Redirector Client
and VSAM logic 49
configuring 49

VSAM Redirector Client *(continued)*
overview 45
possible redirection modes 51
VSAM Redirector Connector
configuring VSAM Redirector Client / VSAM Capture Exit 49
downloading VSAM Redirector Server from Internet 63
installing & implementing 45
installing VSAM Redirector Server 63
obtaining VSAM Redirector Server from PRD1.BASE 63
overview diagram 45
overview diagram of how it works 45
prerequisites for installing 63
VSAM Redirector Handler
calling 66
characteristics of 67
coding VSAM logic 65
CSVFileHandler 67
datatype conversions 67
DB2Handler 67
DB2Handler sample 70
DBHandler 67
error reporting 66
getting a map dynamically 67
IBM-supplied 67
implementing 65
overview 45
overview diagram of where used 45
VSAM Redirector Loader
characteristics of 69
IBM-supplied 69
VSAM Redirector Server
error reporting 66
installing 63
obtaining a copy 63
overview 45
properties file for 65
VSAM-via-CICS service
CICS transactions for use with 101
configuring CICS for 99
description 99
how it works 101
VSAM.RECORD.MAPPING.DEFS 104
VSAM.VSESP.USER.CATALOG 231
VsamDataMapping.java 106
VsamMappingApplet.html 105
VSAMSEL sample Stored Procedure 210
VsamSpaceUsage 192
VSAMSQL CLI environment, de-allocating 213
VSAMSQL CLI environment, initializing 211
VSAMSQL prefix 398
VSAMSQLBindCol(), example 212
VSAMSQLBindParameter(), example 212
VSAMSQLCloseTable - close a specified table (cluster) 399
VSAMSQLExecute(), example 212
VSAMSQLFetch(), example 212
VSAMSQLPrepare(), example 212
VSE Connector Client
description 23
VSE Connector Client *(continued)*
downloading from Internet 25, 75
installing 24, 75
obtaining from PRD2.PROD 25, 75
prerequisites for installing 25, 75
uninstalling 27, 76
VSE Connector Server
action codes supported by 282
and security 36
configuring on the z/VSE host 27
date-format (configuring) 33
entering a command for 35
glossary description 470
job SKVCSCAT (catalog members) 28
job SKVCSSTJ (placing startup job in reader queue) 28
listing possible commands for 35
overview 24
skeleton SKVCSCFG (specify general settings) 29
skeleton SKVCSLIB (specify libraries) 31
skeleton SKVCSPLG (specify plugins) 31
skeleton SKVCSSSL (configure for SSL) 32
skeleton SKVCSUSR (specify logon access) 32
starting 34
testing communication with VSE Connector Client 34
VSE data
accessing using applets 187
accessing using JSPs 245
accessing using servlets 227
representing using EJBs 249
VSE Health Checker
description 175
installing 177
migrating from earlier versions 176
prerequisites 176
starting 177
VSE HTTP Server
configuring 19
VSE HTTP Server, configuring 19
VSE Java Beans
accessing DL/I data using 159
accessing ICCF data using 168
accessing Librarian using 165
accessing operator console using 155
accessing POWER data using 163
accessing VSAM data using 156
and VSE Health Checker application 175
and VSE Navigator application 170
contents of class library 143
glossary description 470
how compare to EJBs 142
how compare to JavaBeans 142
submitting jobs using 152
using for connecting to z/VSE host 148
using for connecting to z/VSE host via SSL 149
using in 3-tier env. 14
using in 3-tier environments 142
using the callback mechanism 146
VSE Java Beans *(continued)*
what they are 142
VSE monitoring agent
collecting data via JMibBrowser 121
collecting data via SNMP command-line tools 121
commands available 120
configuring 117
configuring monitoring plug-ins 119
creating own monitoring plug-ins 127
diagram 115
OIDs (Object Identifiers) 119
overview 115
replacing the configuration file 117
sample configuration file 117
security 118
starting via SKSTMAS job 117
Trap Client API, return codes 127
Trap COBOL and PL/I Batch interface, copybook for 126
Trap LE/C interface, functions provided by 124
using SNMP Trap Client in a batch job 121
using Trap Client API (SNMP) in CICS programs 125
using Trap Client API (SNMP) in COBOL and PL/I programs 124
using Trap Client API (SNMP) in LE/C programs 123
VSE Navigator
and the PowerGridLayout class 196
description 170
VSE Script Client
description 39
example 445
running as a batch program 457
running as a CICS client 459
sample files you can use 445
SSL parameter 455
using symbolic parameters in a batch program 457
using to obtain data 455
VSE Script connector
diagram of how it is used 437
overview 39
protocol used between client and server 439
VSE Script Language 440
VSE Script Language
built-in console functions 443
built-in functions 441
built-in Librarian functions 443
built-in POWER functions 443
built-in string functions 442
built-in VSAM functions 444
description 440
general rules 440
VSE Script sample (getdata.src) 447
VSE Script Server
description 39
installing 39
obtaining from PRD1.BASE 40
obtaining via the internet 40
performing the installation 41
prerequisites for installing 40

VSE Script Server *(continued)*
properties file Connections 43
properties file VSEScriptServer 42
starting locally 449
VSE Scripts, writing 440
VSE Security Manager 36
VSE/POWER data
accessing using VSE Java Beans 163
VSEAppletServer, how used 191
VSECertificateEvent 143
VSECertificateListener 143
VSEConnectionManager 143
VSEConnectionSpec 230
VSEConnectionSpec class 143
VSEConnectorTrace 143
VSEConsole class 143
VSEConsoleExplanation class 143
VSEConsoleMessage class 143
VSEDli class 143
VSEDliPcb class 143
VSEDliPsb class 143
VSEIccf class 143
VSEIccfLibrary class 143
VSEIccfMember class 143
VSELibrarian class 143
VSELibrary class 143
VSELibraryExtent class 143
VSELibraryMember class 143
VSEMessage class 143
VSEPlugin class 143
VSEPower class 143
VSEPowerEntry class 143
VSEPowerQueue class 143
VSEResource class 143
VSEResourceEvent class 143
VSEResourceListener class 143
VSEResourceListener, example 146
VSEScriptServer.properties 42, 446
VSESubLibrary class 143
VSESystem class 143
VSEUser class 143
VSEVsam class 143
VSEVsamCatalog class 143
VSEVsamCluster class 143
VSEVsamField class 143
VSEVsamFilter class 143
VSEVsamMap class 143
VSEVsamRecord class 143
VSEVsamView class 143

W

Web Servers
Apache 20
IBM HTTP Server 20
Lotus Domino Go 20
WebSphere Application Server
and EJB containers 250
complementary programs 6
glossary description 470
installing on Linux on System z 21
installing on Windows, AIX, Sun Solaris 21
installing on z/OS 21
Internet address 6
managing EJBs 249
overview 6
WebSphere Application Server *(continued)*
standard, advanced, & enterprise editions 21
storing session information 230
used for accessing VSAM data 395
WebSphere MQ connectivity
using in 3-tier env. 14
what it offers 6

X

XML parser 418, 420

Z

z/VSE host database, establishing connection to 209

Readers' Comments — We'd Like to Hear from You

IBM z/VSE
e-business Connectors, User's Guide
Version 5 Release 1

Publication No. SC34-2629-02

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Submit your comments using one of these channels:
- Send your comments to the address on the reverse side of this form.
- Send a fax to the following number: +49-7031-163456
- Send your comments via email to: s390id@de.ibm.com
- Send a note from the web page: http://www.ibm.com/systems/z/os/zvse/

If you would like a response from IBM, please fill in the following information:

Name	Address
Company or Organization	
Phone No.	Email address

Readers' Comments — We'd Like to Hear from You
SC34-2629-02

Cut or Fold
Along Line

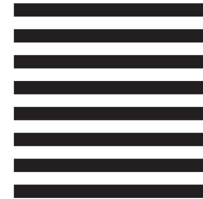
Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE



IBM Deutschland Research & Development GmbH
Department 3248
Schoenaicher Strasse 220
71032 Boeblingen
Germany

Fold and Tape | **Please do not staple** | Fold and Tape

SC34-2629-02

Cut or Fold
Along Line

Product Number: 5609-ZV5

Printed in USA

SC34-2629-02