FIDELITY NATIONAL INFORMATION SERVICES

# FIS Profile

# JDBC Driver Version 3.0.1 User Manual

Published November 2006

# Table of Contents

# About This Manual

The purpose of the **Fidelity JDBC Driver User Manual** is to provide information needed to use and maintain the Java DataBase Connectivity (JDBC) driver in a Profile® user environment. The primary target audience is customer developers who write Java programs to submit to the JDBC Driver to view and maintain data. Topics in this document include:

| | |
|---|---|
| **Chapter 1** | The **JDBC Driver Overview** chapter provides general overview information on what the JDBC Driver is and how it is used. |
| **Chapter 2** | The **Using JDBC Driver** chapter provides information on the basic installation procedure and its connection, stored procedure calls, and Profile service classes. |
| **Appendix A** | The **JDBC API Test Program** appendix provides the lines of code for a program you can use to test the JDBC Application Programming Interface (API). |
| **Appendix B** | The **Multiple MTMs Connection Test Code** appendix provides the lines of code that you can use to test the connection to multiple MTMs using the `ScJDBCConnectionPoolCache` class. |
| **Appendix C** | The **Connection Pooling Parent Class** appendix provides additional information on how the classes used by the Fidelity JDBC Driver were derived (parent class) as used in relation to connection pooling. |
| **Appendix D** | The **Data Type Mappings** appendix shows the mapping between Database data type and JDBC data type. |
| **Appendix E** | The Limitations appendix lists the known limitations that users need to be aware of regarding the Profile JDBC Driver capabilities. |

When viewing this document online, you can click the underlined hyperlinks to view related information (e.g., click the section names above to jump to that section of text).

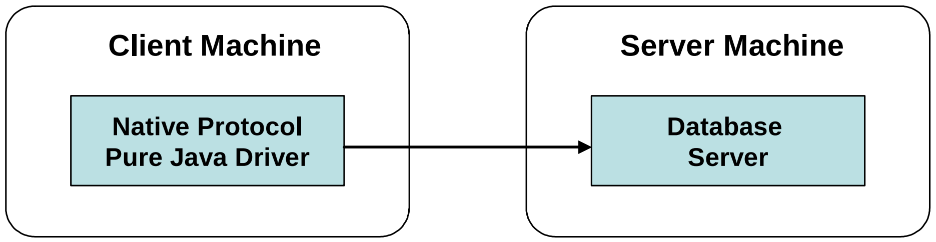
# Chapter 1. JDBC Driver Overview

This chapter provides a general overview of Java DataBase Connectivity (JDBC). For more detailed information, access the JDBC Technology page of the Java Sun Developer Network web site (http://java.sun.com/products/jdbc/).

## What is a JDBC Driver?

JDBC is a Java Application Programming Interface (API) for connecting programs written in Java to a wide range of databases. JDBC enables you to write a single application that can send Structured Query Language (SQL) statements to different Database Management Systems (DBMS) that have a JDBC Driver.

The JDBC Driver acts as a middle-tier layer and enables the connection and transfer of data between the JDBC API and the different databases through Java programs that execute SQL statements. The following figure illustrates the relationship between client and server machines using a JDBC Driver.



## JDBC Specifications and Standards

JDBC is an API developed by Sun Microsystems that provides a standard way to access the data in a database.

**The Fidelity Profile JDBC Driver supports all core JDBC Version 2.0 specifications. The extended JDBC 3.0 APIs are not supported.**

**The Profile JDBC Driver supports all FIS Profile V5.x, V6.x, and V7.x releases. It does not support FIS Profile V4.x releases.**

The JDBC 2.0 API supports the following features:

- **Result Set Enhancements** – Two new capabilities added to Result Sets are Scrolling and Updatability. Scrolling is the ability to move backward and forward through the Result Set contents. Result sets that support this capability are called **Scrollable Result Sets.** Updatability refers to the ability to update the contents of a Result Set and copy the changes to the database.

- **Batch Updates** – A Batch Update is a set of multiple update statements that are submitted to the database for processing as a batch. Sending multiple update statements to the database is much more efficient than sending each update statement separately.

- **Connection Pooling** – Establishing JDBC connections is resource-expensive, especially when the JDBC API is used in a middle-tier server environment. Performance can be improved significantly by using connection pooling. Connection pooling means that connections are reused rather than created each time a connection is requested. The top layer is based on the JDBC Connection Pool Data Source implementation. This enables a single connection cache. The driver supports different transport types (such as MTM, a socket-based transport, and MQ Series transport) and is independent of messaging formats.

- **Rowset** – The `sun.jdbc.rowset` package is used to support the rowset features.

The Profile JDBC Driver implementation is based on Java Development Kit (JDK) 1.4.2. It is a Level 3 native implementation.

The Profile JDBC Driver is compliant with JDBC 3.0 standards. However, some of the JDBC 3.0 features are not supported by the Fidelity Profile JDBC Driver because of database limitations. Refer to **Appendix E** for a list of the non-supported features.

## Result Sets

The JDBC 2.0 driver enables you to scroll both forward (first-to-last) and backward (last-to-first) through the contents of a Result Set. In addition, scrollable Result Sets enable relative and absolute positioning. Absolute positioning is the ability to move directly to a row by specifying its absolute position in the Result Set, while relative positioning provides the ability to move to a row by specifying a position that is relative to the current row. For example, you can move directly to the 10th row in a scrollable Result Set, or to the 5th row following the current row. These Result Sets can be updated as well.

# Chapter 2. Using JDBC Driver

This chapter takes you step-by-step through the process of setting up the JDBC Driver and changing its connection properties.

## Setting up the JDBC Driver at a Customer Site

The following steps describe the necessary JDBC-related settings and connection properties for Windows and UNIX environments.

### Prerequisites

Ensure the following setup requirements are completed before installing the JDBC Driver:

1. Install **JDK 1.4.2** on your machine.

   The JDBC Driver cannot be installed unless JDK 1.4.2 is installed. Also ensure that JDK 1.4.2 has the correct CLASSPATH setting.

2. Install **Fidelity Profile** V5.x or later.

3. The computer on which the JDBC Driver is running must have a network card installed and must be connected to a TCP/IP network to communicate with the database servers.

### Installation Instructions

You must install the JDBC Driver to access the database system. To enable this, store the **ScJDBC.jar** file where it can be found by the Java Virtual Machine (JVM).

To install the JDBC Driver:

1. **Do not unzip** the JDBC Driver **ScJDBC.jar** file. Java recognizes the class files from the jar files.

2. Copy the JDBC Driver **ScJDBC.jar** files to the appropriate installation directory. For example, set the default directory to D:\Sanchez.

3. Add the complete path of the **ScJDBC.jar** file to the environment variable **CLASSPATH**.

4. Enter the following, according to your operating system:

   - On Microsoft Windows:

     ```
     SET CLASSPATH=%CLASSPATH%; D:\sanchez\ScJDBC.jar
     ```

   - On UNIX:

     Enter the following lines in the **.profile** file:

```
        Export runjre=/opt/java/bin/jre
        Export CLASSPATH=/home/user/ScJDBC.jar
```

## Loading the Driver Interface

The Java method used to load all JDBC Drivers is the `Class.forName` method. When you load a driver class, it creates an instance of itself and registers it with the Driver Manager. You can load and register a driver by entering the following call:

```
    Class.forName("sanchez.jdbc.driver.ScDriver");
```

This loads the JDBC Driver and registers it with the JDBC Driver Manager. Once you have loaded and registered a driver with the Driver Manager, it is ready to be used to connect to a database.

## Making the Connection

To establish a connection, enter the appropriate driver connection to the DBMS using any of the following syntax:

```
DriverManager.getConnection(String URL)
DriverManager.getConnection(String URL, java.util.Properties info)
DriverManager.getConnection(String URL, String user, String password);
```

Log on to your DBMS with the login name "admin" and a password of "pass". The URL syntax is given below:

```
String URL = “protocol=jdbc:sanchez/database=140.140.1.1:18610:SCA$IBS/user=admin/password=pass”;
Connection con = DriverManager.getConnection(URL);
```

Or

```
String URL = "protocol=jdbc:sanchez:/database=140.140.1.1:18610:SCA$IBS";
Connection con = DriverManager.getConnection (URL, "admin", "pass");
```

## Changing Connection Properties

You can change the connection properties of the JDBC Driver to affect client-side connection behavior. The following table shows a set of properties that can be used to change the connection properties.

| **Property Name** | **Req** | **Valid Entries** | **Default Value** | **Description** |
|---|---|---|---|---|
| protocol | Y | jdbc:sanchez | | Driver protocol |
| database | Y | <IP Address>:<Port:>:<Server Type> | | **IP Address** - targeted server IP address<br>**Port** - targeted server port number<br>**Server Type** - Profile server type |

| Property Name | Req | Valid Entries | Default Value | Description |
|---|---|---|---|---|
| transType | N | MTM or MQ | MTM | Transaction type, which is independent of message type.<br><br>The basic transports currently supported are:<br><br>■ **MTM** (lightweight socket-based Message Transport Manager)<br>■ **MQ** Series (IBM) |
| locale | N | <Country Code>:<br><Language Code> | US:EN | The Java standard means of identifying a language for the purposes of internationalization and localization.<br>**Country Code**: a non-null string containing a two-letter ISO-639 country code.<br>**Language Code**: a non-null string containing a two-letter ISO-639 language code. |
| dateFormat | N | Pattern | yyyy-mm-dd | The pattern format to use for the date string. |
| timeFormat | N | Pattern | hh:mm:ss | The pattern format to use for the time string. |
| rowPrefetch | N | Row | 30 | Estimated number of rows to fetch from the database when more rows are needed for this Result Set. |
| signOnType | N | 0 or 1 | 0 | **0** – Sign on at clear password.<br><br>**1** – Sign on at challenge/response by encrypted password. |
| numberFormat | N | Pattern | ####.## | The pattern format to use for the number string. |
| poolSize | N | <Number> | 80 | Connection pool size for the connection pool tokens. |

| Property Name | Req | Valid Entries | Default Value | Description |
|---|---|---|---|---|
| fileEncoding | N | ISO-8859-2<br>or<br>ISO-8859-1 | ISO-8859-1 | Character set to be supported:<br><br>**ISO-8859-2** (Latin 2) contains encoded fonts used in many Central and Eastern European languages.<br><br>**ISO-8859-1** (Latin 1) contains standard character encoding of the Latin alphabet. Used for English and many Western European languages. |
| timestampFormat | N | Pattern | yyyy-mm-ddhh:mm:ss.fffffffff | The pattern format to use for the timestamp string. |
| usesp | N | y or yes | N | Enter **y** or **yes** to use stored procedures for a SQL query. |
| garbageCollect | N | 0 or 1 | 0 | **0** - Invoke the garbage collector from the JDBC Driver.<br>**1** – Do not invoke the garbage collector from the JDBC Driver. |
| processMRPC | N | 0 or 1 | 1 | **0** – Do not use custom processing of specific M Remote Procedure Calls (MRPCs) inside the JDBC Driver.<br>**1** – Use custom processing of specific MRPCs inside the JDBC Driver. |
| newPWD | N | <String> | null | **New Password for the user ID.** |
| instID | N | <String> | null | **Institution ID to be sent along with the Login information.** |

## Example

The following example illustrates the loading and connecting of a JDBC Driver by using the properties described in the Connection Properties table.

```
// load the jdbc driver
Class.forName ("sanchez.jdbc.driver.ScDriver");
//create URL
String URL = "protocol=jdbc:sanchez:/database=140.140.1.1:18610:SCA$IBS";
//connect to the driver
Connection con = DriverManager.getConnection (URL, "admin", "pass");
```

For the driver to function properly, you must include all the required attributes. The Profile JDBC Driver allows the following characters to be used as the delimiters of a URL:

| **Valid Delimiters** |
|---|
| **/**   **/ /**   **\|** |

The following example shows the code that you can use to change the date format. In this example, the standard delimiter is **|** and the date delimiter is **/**.

```
String url =
“protocol=jdbc:sanchez|database=140.140.1.1:18610:SCA$IBS|dateFormat=mm/dd/yyyy|fileEncoding=Cp12
52”
```

## Connecting Through the Physical Connection Pool

You can use the connection pool available within the JDBC Driver to pool physical connections to the Profile server. The following sample code illustrates how the connection pool can be used to connect to the same MTM. For additional information on how the classes used by the Fidelity JDBC Driver were derived (parent class), see **Appendix C**.

In the code examples in this document, two consecutive lines that have just a dot (.) on the line indicate locations where customer-specific code should be placed.

## JDBC Connection Using a Connection Pool

```
import sanchez.jdbc.pool.ScConnectionPoolDataSource;
import sanchez.jdbc.pool.ScJDBCConnectionPoolCache;
import sanchez.jdbc.pool.ScJdbcPool;
.
.
public class Application extends HttpServlet {
.
.
            private static ScJDBCConnectionPoolCache ds;
.
.
public void init(ServletConfig config) throws ServletException {
.
.
    try {

        ScConnectionPoolDataSource conPoolDataSource = new
        ScConnectionPoolDataSource();
        conPoolDataSource.setUser(user);  //Set user ID
        conPoolDataSource.setPassword(pass); //Set password
        ScJdbcPool connectionPool = new ScJdbcPool(user, pass, durl);

        connectionPool.setMaximumSize(maximumConnections);   connectionPool.setMinimumSize(minimu
        mConnections);
        ds = new ScJDBCConnectionPoolCache(conPoolDataSource);
        ds.setConCache(connectionPool);  //Set the Cache Pool
        ds.setUser(user);   // Set user ID for the datasource
        ds.setPassword(pass);  //Set password for the datasource

    } catch (SQLException e) {
             Logger.fatal("SQLException in Application.init() ", e);

       }
     .
     .
     }
.
.
  ds.getConnection();  //Get the connection
.
.
}
```

You can use `ds.close()` to close the connections.

## Creating Pools for two Different MTMs from the Same Program

The JDBC Driver enables you to connect to two different MTMs from the same connection pool. The following sample code illustrates how the connection pool can be used to connect to multiple MTMs.

```
//Pool for MTM one
sanchez.jdbc.pool.ScConnectionPoolDataSource ds = new
sanchez.jdbc.pool.ScConnectionPoolDataSource();
String url1 ="protocol=jdbc:sanchez/database=140.140.1.215:18386:
SCA$MKT";
ds.setURL(url1);
Connection con = null;
Statement statement = null;
ResultSet rs = null;
try{
.
.
    con = ds.getConnection("1","xxx");
     statement = con.createStatement();
.
.
}
catch (SQLException ex) {
       ex.printStackTrace();
}
finally{
         if (rs!=null) rs.close();
         if (statement!=null) statement.close();
         if (con!=null) con.close();
}
//Pool for MTM two
sanchez.jdbc.pool.ScConnectionPoolDataSource ds1 = new
sanchez.jdbc.pool.ScConnectionPoolDataSource();
String url2 ="protocol=jdbc:sanchez/database=140.140.1.215:18326:SCA$IBS";
ds1.setURL(url2);

Connection con1 = null;
Statement statement1 = null;
ResultSet rs1 = null;
try{
con1 = ds1.getConnection("1","xxx");
statement1 = con1.createStatement();
.
.
}
catch (SQLException ex) {
ex.printStackTrace();
}
finally{
if (rs1!=null) rs1.close();
if (statement1!=null) statement1.close();
if (con1!=null) con1.close();
 }
```

## Stored Procedure Calls and Profile Service Classes

A stored procedure is a group of SQL statements that performs a specific task or queries to execute on a database server.

You can access the MSQL server class and PROFILE/SQL service class by using the JDBC Statement and PreparedStatement API. You can access the other server classes such as MRPC, TSSP, and NMSP by the JDBC stored procedure API `CallableStatement` using the following syntax:

```
ResultSet rs = {call procedure_name(?,?,….)}
```

## Setting Parameters

Standard Java `CallableStatement` objects can take three types of parameters: IN, OUT, and INOUT. However, the Fidelity JDBC Driver only supports IN and OUT parameters.

### IN Parameters

You can pass in any IN parameter values to a `CallableStatement` object using the `setXXX` methods inherited from `PreparedStatement`. The type of value passed in determines which `setXXX` method is used (e.g., `setFloat` to pass in a float value, `setBoolean` to pass in a Boolean).

### OUT Parameters

If the server class returns OUT parameters, the JDBC type of each OUT parameter must be registered before the `CallableStatement` object can be executed. This is necessary because the DBMS requires the SQL type. Register the JDBC type using the `registerOutParameter` method.

After the statement executes, the `CallableStatement's` `getXXX` method is used to retrieve OUT parameter values. The correct `CallableStatement.getXXX` method to use is determined by the Java type that corresponds to the JDBC type registered for that parameter. For example, `registerOutParameter` uses a JDBC type and `getXXX` method casts this to a Java type.

When a method uses `int` to specify which parameters can be used to act on the `setXXX`, `getXXX`, and `registerOutParameter`, the `int` refers to the '?' placeholder parameters only with numbering starting at one. The parameter number does not refer to literal parameters that are supplied to a stored procedure call. For example, the following code fragment illustrates a stored procedure call with one literal parameter and one '?' parameter:

```
CallableStatement cstmt = con.prepareCall("{call MRPC(1000,?)};
cstmt.registerOutParameter(1,java.sql.Types.TINYINT);
```

In this code, the first argument to `registerOutParameter` - the integer 1 - refers to the first '?' parameter (and in this case, the only '?' parameter). It does not refer to the literal 1000, which is the first parameter to the stored procedure.

## Types of Profile Service Classes

Profile supports a set of service classes categorized by the type of specific activity the client requests. The service classes are provided for financial transactions (TSSP), M code Remote Procedure Calls (MRPC), SQL access (PROFILE/SQL), and Network management functions (NMSP).

The JDBC Driver can access server classes of FIS Profile, MSQL, MRPC, TSSP, HTML, and XML by calling the corresponding JDBC APIs.

The following table describes the Profile Service Classes:

| **Service Class** | **Description** |
|---|---|
| MSQL | The PROFILE/MSQL service class provides a means to interact with the Profile database using an extensive subset of standard SQL statements. Major SQL functionality is provided, with additional PROFILE/SQL-specific extensions and qualifiers available to provide a broader range of control than that provided simply through standard SQL. |
| MRPC | The M Remote Procedure Calls service class provides a mechanism for a client to run public M procedural code on the Profile host. Client applications can run procedures on the host for program-level functions that cannot be performed on the client. This includes such procedures as callbacks to calculate accrual adjustments, or to obtain complex index rates, or the record exchange capability. |
| TSSP | The TSSP service class handles client-initiated requests for debit and credit transactions to financial accounts (e.g., deposit, loan, and general ledger accounts) on the host system. This service class enables clients to perform transactions against Profile accounts. |
| HTML | The HTML service class provides client access to Profile functionality from a web browser over an Intranet or the Internet.<br>■ This service class takes advantage of a development specification for mapping data from tables within Profile to fields on a web page.<br>■ This service class converts original HTML to a run-time procedure that encapsulates SQL to perform data access. The HTML page catalog stores procedure names to be referenced when the pages are served.<br>■ The HTML service class parses named pairs from a form and declares local variables in Profile. A previously compiled procedure, stored on the Profile host, executes at run-time to produce dynamic HTML pages to present on the browser. |
| XML | The XML service class provides client access to Profile functionality from a web-based browser over an Intranet or the Internet. |
| NMSP | The Network Management Stored Procedures (NMSP) service class provides network management transactions control access to the Profile host system and specific control functions. The NMSP service class manages the following client/server connections: client sign-on, client sign-off, and heartbeat message. |

| Service Class | Description |
|---|---|
| AGGMSG | The Aggregate Message (AGGMSG) service class enables the MSQL, MRPC, and TSSP service class calls to bundle together and execute as a single call. An aggregate message acts as a container to present a set of messages at one time to the Profile Banking Server. |

## MRPC Call Syntax

The JDBC Driver supports the MRPC service classes. The `CallableStatement` is invoked by the following syntax:

```
CallableStatement cstmt = con.PrepareCall({call MRPC(MRPCId,?,?,…)});
ResultSet rs = cstmt.executeQuery ( );
```

The “?” placeholders are IN and OUT parameters depending on the stored procedure MRPC.

The passing in of any IN parameter values to a callable statement object is performed using the `setXXX` method. For a particular MRPC call, `setObject` will accept input for a parameter at run time.

By default, the version of the MRPC is assumed as 1. To change the version of the called MRPC, use the following syntax:

```
CallableStatement cstmt = con.PrepareCall({call xmrpc(MRPCId,version,?,?,…)});
ResultSet rs = cstmt.executeQuery ( );
```

MRPC can provide supervisory authorizations when it is called. To specify the supervisory authorization in the MRPC call, enter one of the following in the last parameter of the MRPC call:

- `SPV_AUTH=1 (automatic supervisory authorization for this particular transaction)`
- `SPV_AUTH=0 (no supervisory authorizations)`
- `SPV_AUTH=* (automatic authorization for all transactions specified in the operation)`

## Examples

The following examples further illustrate the JDBC concepts and usage.

Following are the input and output parameters that are applicable to the examples:

**Input:**

```
Beginning Date= "01/22/1997"; Ending Date = "11/26/1997"; Include Final Date Flag="1"; Principal
Amount="1000"; Interest Accrual Method="01"; Interest Rate="12"; Interest Frequency="1MA1"; Next
Interest Date="02/01/1997";     Last Interest Date ="01/01/1997"
```

**Output:**

```
Column Name = "INTAMT";
Column Type = “String”
```

## *Method 1 Example*

This example illustrates how to call MRPC046 (interest calculation utility) by using the Callable Statement API.

```
CallableStatement cstatmt = con.prepareCall("{call mrpc(46,?,?)}");

String sParam=
{ScUtility.JulianDate("11/26/1962"),ScUtility.JulianDate("11/26/1997"),"1","1000","01","12","1MA1
",ScUtility.JulianDate("02/01/1997"),ScUtility .JulianDate("01/01/1997")};
cstatmt.setObject(1, sParam);
cstatmt.registerOutParameter(2, 12, "INTAMT");
ResultSet rs = cstatmt.executeQuery();
System.out.print("\n INTAMT:"+cstatmt.getString(2));
```

## *Method 2 Example*

This example illustrates another way to call MRPC046 by using the Callable Statement API.

```
CallableStatement cstatmt = con.prepareCall("{call mrpc(46,?,?,?,?,?,?,?,?,?,?)}");
cstatmt.setDate(1, new java.sql.Date(1962,11-1, 26));
cstatmt.setDate(2, new java.sql.Date(1997,11-1,26));
cstatmt.setString(3, "1");
cstatmt.setString(4, "1000");
cstatmt.setString(5, "01");
cstatmt.setString(6, "12");
cstatmt.setString(7, "1MA1");
cstatmt.setDate(8, new java.sql.Date(1997,2-1,1));
cstatmt.setDate(9, new java.sql.Date(1997,1-1,1));
cstatmt.setString(8, "57010");
cstatmt.setString(9, "56979");
cstatmt.registerOutParameter(10, 12, "INTAMT");
ResultSet  rs = cstatmt.executeQuery();
```

## *Method 3 (XBAD Calls) Example*

The example shows how to call MRPC043 (account creation) using the Callable Statement API, and illustrates the use of supervisory authorizations.

```
CallableStatement cstatmt = con.prepareCall("{call mrpc(43,?,?,?,?,?,?,SPV_AUTH=1)}");
String ACN=”11872”;
String CID=”957160407215”;
String sParam= “DEP.CID=”+CID+”,DEP.ODO=3,DEP.ACNRELC=A,DEP.ACCTNAME=My Account,
CMBSTM1.STMGRP=1,RELCIF1.ACN=”+ACN+”,RELCIF1.ROLE=1”;
cstatmt.setString(1,”400”);
cstatmt.setString(2,”USD”);
cstatmt.setString(3, sParam);
cstatmt.setString(4,”1”);
cstatmt.setString(5,”0”);
cstatmt.registerOutParameter(6, 12, "RETACN");
ResultSet rs = cstatmt.executeQuery();
```

## *HTML Call Syntax*

The HTML service class provides the client access to Profile functionality. The Callable Statement is invoked by the following syntax:

```
CallableStatement cstmt  = {call html(?,?)}
setString(1,java.sql. VARCHAR)
registerOutParameter(2, java.sql. VARCHAR)
ResultSet rs = cstatmt.executeQuery()
```

## *XML Call Syntax*

The XML service class provides the client access to Profile functionality. The Callable Statement is invoked by the following syntax:

```
CallableStatement cstmt  = {call XML(?,?)}
setString(1,java.sql. VARCHAR)
registerOutParameter(2, java.sql. VARCHAR)
ResultSet rs = cstatmt.executeQuery()
```

## *TSSP Call Syntax:*

The TSSP service class handles client-initiated requests for debit and credit transactions to financial accounts on the host system and the Callable Statement is invoked by the following syntax:

```
CallableStatement cstmt = {call tssp(?,?)}
setObject(1,(ScTsspRecord[])tsspRequestArray)
registerOutParameter(2, -2,”ResultSet”)
ResultSet rs = cstatmt.executeQuery()
```

## *Construction of ScTsspRecord:*

```
public  ScTsspRecord(String BRCD, String TPD, String ACN, String ETC, String TAMT, String TRC,
               String[] CRCD*, String EFD, String VDT, String[] SPV*, String[] QLF*,
               String[] PSI*, String[] MSC*, String TXMT, String HISTN,String[] CKREG*)
```

The following table describes the ScTsspRecord input parameters and the data fields included in the standard message. Data fields that are complex in nature (e.g., may contain multiple pieces of information) are designated with a “*” next to the field position number.

| POS | Name | Req | Description | Type | Max Len |
|---|---|---|---|---|---|
| 1 | BRCD | Y | Branch Code<br>TABLE=[UTBLBRCD]<br>Conditionally required.<br>DEFAULT=first transaction’s BRCD field | N | 6 |
| 2 | TPD | N | Teller Posting Date<br>DEFAULT=[CUVAR]TJD (current system date) for first transaction; first transaction’s TPD for remaining transactions. | D | 5 |
| 3 | ACN | Y | Account Number<br>Identifies the customer account or general ledger number. | N | 12 |
| 4 | ETC | Y | Transaction Code<br>TABLE=[TRN] | T | 12 |
| 5 | TAMT | Y | Transaction amount | $ | 14 |

| POS | Name | Req | Description | Type | Max Len |
|---|---|---|---|---|---|
| 6 | TRC | Y | Transaction Reference Number<br><br>Identifies the transaction as it is stored in the client's electronic journal. | N | 12 |
| 7* | CRCD | N | Currency Code<br><br>This complex field contains the currency of the transaction, the exchanged currency, and the exchange rate. | T | 500 |
| 8 | EFD | N | Effective Date<br><br>The date when the principal amount of the transaction takes effect for general ledger purposes.<br><br>DEFAULT=[CUVAR]TJD | D | 5 |
| 9 | YDT | N | Value Date<br><br>The date when the transaction takes effect for accrual purposes.<br><br>DEFAULT=[CUVAR]TJD | D | 5 |
| 10* | SPV | N | Supervisory Authorization Information | T | 500 |
| 11* | QLF | N | Transaction Qualifiers<br><br>TABLE=[STBLQLF] | T | 500 |
| 12* | PSI | N | Payment System Instructions<br><br>TABLE=[STBLPSI] | T | 500 |
| 13* | MSC | N | Miscellaneous Information Codes<br><br>TABLE=[STBLMSC] | T | 500 |
| 14 | TCMT | N | Free Text Transaction Comment | T | 40 |
| 15 | HISTN | N | Transaction Notes | M | 2000 |
| 16* | CKREG | N | Check Register Information | T | 500 |

**Key:**
***** denotes complex fields
**Type** codes: **N**=Numeric, **D**=Date, **T**=Text, **$**=Currency, **M**=Memo

Instead of using the custom ScTsspRecord for the input data, the same information can be sent using a Hashtable using the above fields as keys.

Refer to **Appendix B** for an example of a TSSP server class called by a `CallableStatement`.

For detailed information about TSSP, refer to the *Transaction Services and Stored Procedures (TSSP)* section of the **Profile Enterprise Server Architecture** document.

## TSSP Error Handling

The transactions can return multiple error messages or authorization messages. All the messages from Profile will be forwarded to the client application and each message will be tagged with its own transaction ID, to identify which messages belong to which transactions.

## AGGMSG Call Syntax

The Aggregate Message service class enables the system to bundle together the SQL, MRPC, and TSSP calls and submit them as a single call to the database. The `CallableStatement` is used to call the AGGMSG and is invoked by the following call statement:

```
CallableStatement cstmt = con.PrepareCall({call AGGMSG(?,?,…)});
ResultSet rs = cstmt.execute();
```

The “?” placeholders are IN and OUT parameters.

Use the `setXXX` method to pass IN parameter values to a callable statement object.

The AGGMSG call is similar to an MRPC call in syntax, but its parameters differ. The AGGMSG call accepts `PreparedStatement` and `CallableStatement` objects as parameters. The results are returned to the calling program as a HashMap. The first parameter to the AGGMSG call is a HashMap, followed by the error handling option (EHO). The error handling option is sent to Profile along with the other calls. The options are:

- **1** – If a sub-record fails, roll back all prior updates, stop processing, and mark previously processed sub-records as rolled-back. This fails the entire set.
- **2** – If a sub-record fails, stop processing, but retain the updates already done and treat those sub-records as successful.
- **3** – If a sub-record fails, mark it failed, but continue processing. This allows some successes and some failures within a set.

Refer to the **Profile Enterprise Architecture** and **CR13176 Aggregate Message Service Class** documents for details on how Profile handles AGGMSG calls.

The EHO is followed by the aggregate message calls. Each call is provided as a pair: a message ID followed by a `PreparedStatement` or `CallableStatement` object. Perform the following steps to invoke an AGGMSG call:

1. Prepare the SELECT/INSERT/UPDATE calls, MRPC, calls and TSSP calls to be executed.
2. Prepare the AGGMSG call.

3. Provide the HashMap, EHO, and `PreparedStatement/CallableStatement` objects using the `setObject` call.

4. Call `execute()` to execute the calls.

5. Process the HashMap for the results.

A fetchNext() **cannot** be performed on the resultsets returned for a SELECT SQL that is executed using the AGGMSG call.

## Example

```
/* Prepare the UPDATE statement using PrepareStatment */
String tempnotes="TEST1";
PreparedStatement upstmt1 = con.prepareStatement("UPDATE LNTRS1 SET PAYDES=? WHERE CID=? AND
PAYID=?");
upstmt1.setString(1,tempnotes);
upstmt1.setString(2,"6");
upstmt1.setString(3,"DFM1");

/* Prepare the SELECT statement using PrepareStatment */
PreparedStatement pstmt = con.prepareStatement("SELECT REMND,INSRT,CID,PAYID,PAYDES,TRTYPE,RAMTP
from lntrs1 where CID=? and PAYID=?");
pstmt.setString(1,"701");
pstmt.setString(2,"NATL");

/* Prepare the MRPC call using CallableStatement */
CallableStatement cstatmt1 = con.prepareCall("{call mrpc(10,?,?,?)}");
cstatmt1.setString(1, "SANCH");
cstatmt1.setString(2, "D");
cstatmt1.registerOutParameter(3, 12, "RETVAL");

/* Prepare the AGGMSG call */
HashMap hm = new HashMap(3);
CallableStatement cst = con.prepareCall("{call AGGMSG(?,?,?,?,?,?,?,?)}");
cst.setObject(1,hm);
cst.setObject(2,"1");
cst.setObject(3,"001");
cst.setObject(4,pstmt);
cst.setObject(5,"002");
cst.setObject(6,upstmt1);
cst.setObject(7,"003");
cst.setObject(8,cstatmt1);

/* Execute the AGGMSG call */
if (cst.execute() == true)
{
        Integer ret;
        Object obj = hm.get("001");
        System.out.println("\n001\n---");
        /* Check if a resultset is returned – for a SELECT/MRPC*/
        if (obj instanceof ResultSet) {
                ResultSet rs1 = (ResultSet)obj;
                dispResultSet (rs1);
        }
        /* Check if a String object is returned – for UPDATE calls*/
        else if (obj instanceof String) {
                String str = obj.toString();
                System.out.println("\nReturn Value - "+str);
        }
        /* Check if an ERROR occurred */
        else if (obj instanceof SQLException) {
                SQLException eSQL = (SQLException)obj;
                System.out.println ("\n*** SQLException caught ***\n");

                while (eSQL != null) {
                        System.out.println ("SQLState: " + eSQL.getSQLState ());
                        System.out.println ("Message:  " + eSQL.getMessage ());
                        System.out.println ("Vendor:   " + eSQL.getErrorCode ());
                        eSQL = eSQL.getNextException ();
                        System.out.println ("");
                }
        }
        else if (obj instanceof Exception) {
                Exception e = (Exception)obj;
                System.out.println ("\n*** Exception caught ***\n");
                e.printStackTrace ();
        }
        else {
                ret = (Integer)obj;
                System.out.println("Return Value - "+ret);
        }

        Object obj1 = hm.get("002");
        System.out.println("\n002\n---");
        if (obj1 instanceof ResultSet) {
                ResultSet rs3 = (ResultSet)obj1;
                dispResultSet (rs3);
        }
        else if (obj1 instanceof String) {
                String str = obj1.toString();
                System.out.println("\nReturn Value - "+str);
        }
        else if (obj1 instanceof SQLException) {
                SQLException eSQL = (SQLException)obj1;
                System.out.println ("\n*** SQLException caught ***\n");

                while (eSQL != null) {
                        System.out.println ("SQLState: " + eSQL.getSQLState ());
                        System.out.println ("Message:  " + eSQL.getMessage ());
                        System.out.println ("Vendor:   " + eSQL.getErrorCode ());
                        eSQL = eSQL.getNextException ();
                        System.out.println ("");
                }
        }
        else if (obj1 instanceof Exception) {
                Exception e = (Exception)obj1;
                System.out.println ("\n*** Exception caught ***\n");

                e.printStackTrace ();
        }
        else {
                ret = (Integer)obj1;
                System.out.println("Return Value - "+ret);
        }

        Object obj2 = hm.get("003");
        System.out.println("\n003\n---");
        if (obj2 instanceof ResultSet) {
                ResultSet rs4 = (ResultSet)obj2;
                dispResultSet (rs4);
        }
        else if (obj2 instanceof String) {
                String str = obj2.toString();
                System.out.println("\nReturn Value - "+str);
        }
        else if (obj2 instanceof SQLException) {
                SQLException eSQL = (SQLException)obj2;
                System.out.println ("\n*** SQLException caught ***\n");

                while (eSQL != null) {
                        System.out.println ("SQLState: " + eSQL.getSQLState ());
                        System.out.println ("Message:  " + eSQL.getMessage ());
                        System.out.println ("Vendor:   " + eSQL.getErrorCode ());
                        eSQL = eSQL.getNextException ();
                        System.out.println ("");
                }
        }
```

```
        else if (obj2 instanceof Exception) {
                Exception e = (Exception)obj2;
                System.out.println ("\n*** Exception caught ***\n");

                e.printStackTrace ();
        }
        else {
                ret = (Integer)obj2;
                System.out.println("Return Value - "+ret);
        }
}
```

### AGGMSG Error Handling

Most errors are returned through the HashMap as a `SQLException` object or as an `Exception` object.

## Submitting Batch Updates

The Batch Update feature enables an application to submit multiple update statements (insert/update/delete) in a single request to the database. This can provide a dramatic increase in performance when a large number of update statements need to be executed.

The example shown below retrieves the Credit Card information for a Company from the database. The `executeQuery` returns a `ResultSet` object containing the results of the query sent to the DBMS.

```
      //preparedStatement example:
            String[] crcd = {"USD","CAD"};
            String[] co = {"SCA","SCA"};

       PreparedStatement pstmt = con.prepareStatement("select
       crcd,coc from crcd"+ "WHERE crcd = ? and co =?");

            for (int j=0;j<crcd.length;j++)
            {
                pstmt.setString(1, crcd[j]);
                pstmt.setString(2, co[j]);
                rs = pstmt.executeQuery();
                dispResultSet (rs);
            }
```

The example shown below updates the Credit Card information for a Company in the database. The return value for `executeUpdate` is an integer that indicates how many rows of a table were updated and are to be printed.

```
pstmt = con.prepareStatement("update crcd set co='SCA' "+
"WHERE crcd = ? and co =?");
for (int j=0;j<crcd.length;j++)
{
    pstmt.setString(1, crcd[j]);
    pstmt.setString(2, co[j]);
    int k = pstmt.executeUpdate();
    System.out.print("\n update row:"+k);
}
```

## Timestamp Support

The current JDBC Timestamp implementation will work correctly with the database data type **VARCHAR2(30)**. If the DB data type is TIMESTAMP, the nanoseconds value in the timestamp will be empty because of size limitation on the DB side.

The default timestamp pattern (“yyyy-mm-dd hh:mm:ss.fffffffff” ) can be overwritten by specifying the timestamp format or locale (<Country Code>:<Language Code>) in the connection property.

## Internationalized Messages

JDBC Driver error messages are externalized to a resource bundle in order to support Internationalization. ResourceBundle is a tool for supporting messages in multiple languages with help of locale. The error messages are specified in the property file in the format of "name=value" syntax.

Currently, all default error messages are available only in English in the `Resource.properties` property file.

If you need localization in another language (for example, French), copy the `Resource.properties` file to a new file (for example, `Resource_fr.properties`) and translate the error messages to the target language (in this example, French).

Sample entries from the `Resource.properties` (English) property file:

```
Invalid_column_index=Invalid column index
Invalid_column_type=Invalid column type {0}
```

Sample entries from the example `Resource_fr.properties` (French) property file:

```
Invalid_column_index=french translation for "Invalid column index"
Invalid_column_type=french translation for "Invalid column type" {0}
```

## BLOB and CLOB Support

The following examples show how to insert and read Binary Large Object (BLOB) and Character Large Object (CLOB) columns in Profile. Note that the JDBC Driver maps BLOB columns to Binary (B) data type columns in Profile, and maps CLOB columns to Memo (M) data type columns.

## BLOB Example

```
Statement sql = con.createStatement();

//insert a BLOB

String s1 = "insert into cifpic (acn) values (1000)";
sql.executeUpdate(s1);

PreparedStatement pstmt = con.prepareStatement("UPDATE cifpic SET PIC=? WHERE ACN=1000");

FileInputStream in = new FileInputStream("E:\\fileName.jpg");
```

```
byte[] buf = new byte[in.available()];
ByteArrayOutputStream bout = new ByteArrayOutputStream();

int num = 0;
while ((num = in.read(buf)) != -1) {
        bout.write(buf, 0, num);
}

ScSerialBlob blob = new ScSerialBlob(buf);
pstmt.setBlob(1,blob);
pstmt.executeUpdate();

//selecting a BLOB column

ResultSet rs = sql.executeQuery("select PIC from cifpic where ACN=1000");
rs.next();
Blob o = rs.getBlob("PIC");

FileOutputStream out = new FileOutputStream("fileName.jpg",true);
byte[] bytes = o.getBytes(0,(int)o.length());
out.write(bytes);
out.flush();
out.close();
```

## CLOB Example

```
Statement sql = con.createStatement();

//insert a CLOB
String s1 = "INSERT INTO tmplnnew (cid,clntid) VALUES (401,1)";
sql.executeUpdate(s1);

PreparedStatement pstmt = con.prepareStatement("UPDATE tmplnnew SET lnnewstr=? WHERE cid=401 and
clntid=1");

String filename = "E:\\Client_Tool_Kits\\JDBC\\memo.txt";

BufferedReader in = new BufferedReader(
new InputStreamReader(new FileInputStream(filename), "UTF8"));

String nextline;
StringBuffer bf = new StringBuffer();
while ((nextline  = in.readLine()) !=null)
{
        bf = bf.append(nextline).append("\n");
}

String s0 = bf.toString();
char[] chars = s0.toCharArray();
ScSerialClob clob = new ScSerialClob(chars);
pstmt.setClob(1,clob);
pstmt.executeUpdate();

//selecting a CLOB column
ResultSet rs = sql.executeQuery("select lnnewstr from tmplnnew where cid=401 and clntid=1");
rs.next();
Clob o = rs.getClob("lnnewstr");
long len = o.length();
String s = o.getSubString(0,(int)len);

BufferedWriter out = new BufferedWriter(
new OutputStreamWriter(new FileOutputStream("clob.txt",true), "UTF8"));
out.write(s.toCharArray());
out.flush();
out.close();
```

## UTF8 and UTF-16 Encoding for Unicode™

Unicode support is being introduced into Profile in the v7.1 release in order to expand the choice of languages in which Profile can be offered (including character sets for Japanese, Chinese, and Korean). Unicode capabilities provide a unified character encoding system within which character sets for all languages can be represented in a consistent manner. Unicode supports a fixed two-byte character representation (UTF-16) and a variable-byte representation (UTF-8). Both forms can be converted between each other without loss of information.

Users can enable UTF-8 or UTF-16 by adding a `fileEncoding` pattern to a URL to get a connection object.

### Example

```
String url = "protocol=jdbc:sanchez/database=10.134.32.156:18326:SCA$IBS/fileEncoding=UTF16";
Connection con = DriverManager.getConnection (url,"userId","password");

//update a column
String s1 = “UPDATE UTFDEMO SET CONTEXT='Chinese: 我能吞下玻璃而不伤身体.我能吞下玻璃而不傷身體,不傷身體!'
WHERE LANGID=6”;
Statement statement = con.createStatement(s1);
statement.executeUpdate();

//selecting a column
String s2 = “select * from utfdemo where context like '%吞下玻璃而不伤身体%'”;
ResultSet rs = statement.executeQuery();
```

## Resultset Holdability Support

A holdable cursor, or result, is one that does not automatically close when the transaction that contains the cursor is committed. JDBC 3.0 adds support for specifying cursor holdability. It is useful feature if a user tries to hold a ResultSet without holding a connection object when connection pooling is used.

### Example

```
String s = "select cid, tseq, tjd, tamt  from hist where cid=300000004462";

Statement statement = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY,
ResultSet.HOLD_CURSORS_OVER_COMMIT);

/*  or use PreparedStatement
PreparedStatement statement =
con.prepareStatement(s,ResultSet.TYPE_SCROLL_INSENSITIVE,ResultSet.CONCUR_READ_ONLY,ResultSet.HOL
D_CURSORS_OVER_COMMIT);
*/

ResultSet rs = statement.executeQuery(s);
con.commit();
statement.close();

//rs is still holdable after con and statement are close.
```

## Retrieving Auto-Generated Keys

Due to a Profile limitation only one key can be auto-generated or retrieved per Insert statement. Furthermore, only the following APIs from the JDBC3.0 specification are supported for this functionality.

```
public boolean execute(String sql, int autoGeneratedKeys)
public int executeUpdate(String sql, int autoGeneratedKeys)
public ResultSet getGeneratedKeys()
```

Refer to the **Limitations** section for a list of unsupported APIs.

### Code Example for Retrieving Auto-Generated Keys:

```
Statement stmt = conn.createStatement();
// indicate that the key generated is going to be returned
int rows = stmt.executeUpdate("INSERT INTO ORDERS (ISBN, CUSTOMERID) " +
"VALUES (195123018, 'BILLG')", Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
boolean b = rs.next();
if (b == true) {
        // retrieve the new key value
        ...
}
```

## Profile Stored Procedure for SQL Query Support

Profile JDBC Driver can be configured to improve the performance of frequently used SQL Queries. Enabling this feature instructs the Profile JDBC Driver to internally convert SQL queries into Stored Procedures.

A new JDBC URL property (**usesp=y** or **usesp=yes**) is added to support Stored Procedures for SQL queries. To invoke the Store Procedure feature for an SQL query, add the usesp=y or usesp=yes pattern to a URL. Once this feature is turned on, the SQL query submitted from the JDBC Prepared Statement interface will internally call the Profile Stored Procedure. This feature is transparent to the user and there is no change in SQL query syntax.

### Example

```
protocol=jdbc:sanchez/database=10.134.32.156:18326:SCA$IBS/usesp=y
```

## JCA and JNDI Support

Profile JDBC Driver is compatible with the J2EE Connector Architecture (JCA) specification and the Java Naming and Directory Interface (JNDI) standards. Users can access all Profile JDBC Driver supported features through the JCA interface. Note that configuring JDBC services for a JCA interface is specific to the J2EE server; please consult your J2EE server manual for configuration information. All JNDI standards should be followed.

### Code Example: Programmatically binding administered object

```
//creating administered ScConnectionPoolDataSource object:
ScConnectionPoolDataSource connPoolDs = new ScConnectionPoolDataSource();
connPoolDs.setURL(url);
connPoolDs.setUser(user);
connPoolDs.setPassword(password);
connPoolDs.setRowPrefetch(10);
connPoolDs.setLoginTimeout(300);

//create context
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.fscontext.RefFSContextFactory");
env.put(Context.PROVIDER_URL, "file:e:/Client_Tool_Kits/jdbcTest/jndi");
Context ctx = new InitialContext(env);
//bind !
ctx.rebind("ProfileConnPoolDs", connPoolDs );
```

## Code Example : Programmatically retrieving administered object

```
//Create context
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.fscontext.RefFSContextFactory");
env.put(Context.PROVIDER_URL, "file:e:/Client_Tool_Kits/jdbcTest/jndi");
Context ctx = new InitialContext(env);
// Read object
ConnectionPoolDataSource conPoolDsRead = (ConnectionPoolDataSource)
ctx.lookup("ProfileConnPoolDs");
ResultSet rs =
conPoolDsRead.getPooledConnection().getConnection().createStatement().executeQuery("select acn
from cif");
//process ResultSet
```

# Appendix A. JDBC API Test Program

This appendix provides the lines of code for a program you can use to test the JDBC API interface.

```
//-----------------------------------------------------------------------
//
// Module: jdbcTest.java
//
// Description: Test program for JDBC API interface. This Java
// application will be connected to the Profile JDBC Driver, issue
// MSQL,MRPC,TSSP statements to FIS Profile and display all results // columns and rows.
// Product:    JDBC Driver
//
// Author: Quansheng Jia
//
// Copyright: Fidelity National Information Services
//-----------------------------------------------------------------------

import java.net.URL;
import java.sql.*;
import sanchez.utils.ScUtility;
import java.io.*;
import java.util.StringTokenizer;
import sanchez.him_pa.formatters.ScTsspRecord;

class jdbcTest {

        public static void main (String args[]) {

                try {

                        // Load the jdbc driver
                        Class.forName ("sanchez.jdbc.driver.ScDriver");
                        // Attempt to connect to a driver. Each one
                        // of the registered drivers will be loaded until
                        // one is found that can process this URL

           //Profile JDBC Driver supports two URL syntaxes
           //syntax one url   = "jdbc:sanchez:thin:@140.140.1.1:18610:SCA$IBS";
       //syntaxtwourl="protocol=jdbc:sanchez/locale=US:EN/database=140.140.1.1:18610:SCA$IBS/tra
nsType=MTM/rowPrefetch=40/fileEncoding=ISO-8859-2/timeOut=2/User=46019/passWord=xxx";

            String url = "protocol=jdbc:sanchez/database=140.140.1.1:18610:SCA$IBS";
            Connection con = DriverManager.getConnection (url, "4609", "xxx");

                        //The driver can be traced on by calling
                        DriverManager.setLogWriter(writer);
                        String errorFile="trace.log";
                        PrintWriter writer = new PrintWriter(new FileWriter(errorFile));
                        DriverManager.setLogWriter(writer);

                        // Check for and display all warnings generated
                        // by the connect.

                        checkForWarning (con.getWarnings ());

                        // Get the DatabaseMetaData object and display
                        // some database information
                        DatabaseMetaData dma = con.getMetaData ();

                        System.out.println("\nConnected to " + dma.getURL());
                        System.out.println("Driver       " +
                                dma.getDriverName());
                        System.out.println("Version      " +
                                dma.getDriverVersion());
                        System.out.println("");
                        System.out.println("Databse Version      " +
                                dma.getDatabaseProductVersion());
                        System.out.println("");

                        ResultSet firstRe;


                        System.out.print("\n dma.getTableTypes():\n");
                        firstRe = dma.getTableTypes();
                        dispResultSet (firstRe);

                        System.out.print("\n dma.getTables:\n");

                        firstRe = dma.getTables(null, null, "CRCD",null);
                        dispResultSet (firstRe);

                        System.out.print("\n dma.getVersionColumns:\n");
                        firstRe = dma.getVersionColumns(null, null, "COL");
                       dispResultSet (firstRe);


                        System.out.print("\n dma.getImportedKeys:\n");
                        firstRe = dma.getImportedKeys(null,null,"LN");
                        dispResultSet (firstRe);

                        System.out.print("\n dma.getCrossReference:\n");
                        firstRe dma.getCrossReference(null,null,"LN",null,null,"ln");
                        dispResultSet (firstRe);


                        System.out.print("\n dma.getPrimaryKeys:\n");
                        firstRe = dma.getPrimaryKeys(null, null, "CRCD");
                        dispResultSet (firstRe);

                        System.out.print("\n dma.getIndexInfo:\n");
                        firstRe = dma.getIndexInfo(null, null, "ACN", false, true);
                        dispResultSet (firstRe);


                        System.out.print("\n dma.getColumns:\n");
                        firstRe = dma.getColumns(null, null, "CRCD",null);
                        dispResultSet (firstRe);


                        // Statement example:
                        Statement stmt = con.createStatement ();

                        //get a picture from host
                        stmt.setFetchSize(1);
                        ResultSet rs = stmt.executeQuery ("select PIC FROM CIFPIC");
                        rs.absolute(1);
                        byte[] picture = rs.getBytes(1);

                        stmt.setFetchSize(50);

                        // Submit a query, creating a ResultSet object
                        rs = stmt.executeQuery ("select crcd.crcd,crcd.co,desc from crcd where
CO='SCA'");
                        rs.absolute(1);
                        System.out.print("\n get CRCD:"+rs.getString("crcd.crcd"));
                        dispResultSet (rs);
                        System.out.print("\n get column CRCD index:"+rs.findColumn("crcd.crcd"));
                        System.out.print("\n get column column name for index
                        2:"+rs.getMetaData().getColumnName(2));

                        Statement stmt1 = con.createStatement();
                        stmt1.executeUpdate("update crcd set desc='United States Dollars' where
crcd='USD' and CO='SCA'");
                        //check if the row updated
                        System.out.print("\n if row update:"+rs.rowUpdated());
```

```
             //preparedStatement example:
            String[] crcd = {"USD","CAD"};
            String[] co = {"SCA","SCA"};

            PreparedStatement pstmt = con.prepareStatement("select
crcd,co,minrate,curdec,desc,cntry from crcd "+
            "WHERE crcd = ? and co =?");
            for (int j=0;j<crcd.length;j++)
            {
                pstmt.setString(1, crcd[j]);
                pstmt.setString(2, co[j]);
                rs = pstmt.executeQuery();
                dispResultSet (rs);
            }

            pstmt = con.prepareStatement("update crcd set co='SCA' "+
            "WHERE crcd = ? and co =?");
            for(int j=0;j<crcd.length;j++)
            {
                pstmt.setString(1, crcd[j]);
                pstmt.setString(2, co[j]);
                int k = pstmt.executeUpdate();
                System.out.print("\n update row:"+k);
            }

        //callableStatement example:

        //mrpc46,interestCalculationUtility
        //input:
                //String BeginningDate= "01/22/1997";
           //String EndingDate = "11/26/1997";
           //String IncludeFinalDateFlag="1";
           //String PrincipalAmount="1000";
           //String InterestAccrualMethod="01";
           //String InterestRate="12";
           //String InterestFrequency="1MA1";
           //String NextInterestDate="02/01/1997";
           //String LastInterestDate ="01/01/1997";
           //output:
           //String columName = {"INTAMT"};
       //String columType = {"T"};
       //String columnLength ={"12"};

               CallableStatement cstatmt = con.prepareCall("{call mrpc(46,?,?)}");
           String[] sParam =
{ScUtility.JulianDate("11/26/1962"),ScUtility.JulianDate("11/26/1997"),"1","1000","01","12","1MA1
",ScUtility.JulianDate("02/01/1997"),ScUtility .JulianDate("01/01/1997")};
            for (int k=0;k<3;k++)
            {
                sParam[5] = String.valueOf((Integer.parseInt(sParam[5])+2));
                cstatmt.setObject(1, sParam);
                cstatmt.registerOutParameter(2, 12, "INTAMT");
                rs = cstatmt.executeQuery();
                dispResultSet (rs);
                System.out.print("\n getString():"+cstatmt.getString(2)+"\n");
            }


            cstatmt = con.prepareCall("{call mrpc(46,?,?,?,?,?,?,?,?,?,?)}");
            cstatmt.setString(1, "44524");
            cstatmt.setString(2, "57308");
            //
            cstatmt.setString(3, "1");
            cstatmt.setString(4, "1000");
            cstatmt.setString(5, "01");
            cstatmt.setString(6, "12");
            cstatmt.setString(7, "1MA1");
            cstatmt.setString(8, "57010");
            cstatmt.setString(9, "56979");
            //
            cstatmt.registerOutParameter(10, 12, "INTAMT");
            rs = cstatmt.executeQuery();
            dispResultSet (rs);

            //TSSP server class example
            tsspServerTest(con);

            //TP example:
            String[] t = new String[6];

                   t[0] ="CREATE TABLE XTEST2(CID INTEGER NOT NULL,BAL number,BOO INTEGER,LNM
VARCHAR(40), primary key (cid)) ";
                   t[1] ="INSERT INTO XTEST2 (CID, BAL, BOO,LNM) VALUES (111,200,10,'SMITH')";
                   t[2] ="UPDATE XTEST2 SET BAL=BAL+100 WHERE CID=111";
                   t[3] ="INSERT INTO XTEST2 (CID,BAL) VALUES (12345,9999)";
                   t[4] ="UPDATE XTEST2 SET BAL=BAL+200 WHERE CID=111";
                   //t[5] ="DROP TABLE XTEST2";
           //if table does not exist, create, or update it.
            try{
               con.setAutoCommit(false);
               stmt.executeUpdate(t[0]);
               stmt.executeUpdate(t[1]);
               stmt.executeUpdate(t[2]);
               stmt.executeUpdate(t[3]);
               con.commit();
            }
            catch (Exception u){
            //executeBatch example:
                con.setAutoCommit(false);
                stmt.addBatch(t[2]);
                stmt.addBatch(t[4]);
                stmt.executeBatch();
            }

            //ResultSet update examples:
            //insert row to result set
            try{
               rs = stmt.executeQuery ("select cid,bal,boo,lnm from xtest2");

               dispResultSet (rs);

               rs.moveToInsertRow();

               rs.updateInt(1, 157);
               rs.updateInt(2, 8881);
               rs.updateInt(3, 10);
               rs.updateString(4, "TEST2");

               rs.beforeFirst();
               dispResultSet (rs);
               rs.insertRow();
            }
            catch (SQLException ee) {System.out.print("\n Error:"+ee.getMessage());}

            //update ResultSet
            rs = stmt.executeQuery ("select cid,bal,boo,lnm from xtest2");
            System.out.print("\n before update\n");
            dispResultSet (rs);
            rs.absolute(2);
            rs.updateInt(2,598888);
            rs.updateString(4,"TEST8");

            System.out.print("\n cancelRowUpdates()\n");
            rs.cancelRowUpdates();
            rs.beforeFirst();
            dispResultSet (rs);

            System.out.print("\n Update a row\n");
            rs.absolute(2);
            rs.updateInt(2,598888);
```

```
    rs.updateString(4,"TEST3");
    rs.updateRow();

    //delete a row from a ResultSet
    rs = stmt.executeQuery ("select cid,bal,boo,lnm from xtest2");
    dispResultSet (rs);

    rs.absolute(2);
    rs.deleteRow();
    rs.beforeFirst();
    dispResultSet(rs);

    System.out.print("\n rs.refreshRow()");
    rs.beforeFirst();
    rs.refreshRow();
    dispResultSet(rs);
        // Close the result set
                rs.close();
                // Close the statement
                stmt.close();
                // Close the connection
                con.close();
        }
        catch (SQLException ex) {

                // A SQLException was generated. Catch it and
                // display the error information. Note that there
                // could be multiple error objects chained
                // together

                System.out.println ("\n*** SQLException caught ***\n");

                while (ex != null) {
                        System.out.println ("SQLState: " +
                                ex.getSQLState ());
                        System.out.println ("Message:  " +
                                ex.getMessage ());
                        System.out.println ("Vendor:   " +
                                ex.getErrorCode ());
                        ex = ex.getNextException ();
                        System.out.println ("");
                }
        }
        catch (java.lang.Exception ex) {

                // Got some other type of exception. Dump it.

                ex.printStackTrace ();
        }

         //SqlFormat(con);
  //System.in.read();
}

//-------------------------------------------------------------------
// checkForWarning
// Checks for and displays warnings. Returns true if a warning
// existed
//-------------------------------------------------------------------

private static boolean checkForWarning (SQLWarning warn)
                throws SQLException
{
        boolean rc = false;

        // If a SQLWarning object was given, display the
        // warning messages. Note that there could be
        // multiple warnings chained together

        if (warn != null) {
                System.out.println ("\n *** Warning ***\n");
                rc = true;
                while (warn != null) {
                        System.out.println ("SQLState: " +
                                warn.getSQLState ());
                        System.out.println ("Message:  " +
                                warn.getMessage ());
                        System.out.println ("Vendor:   " +
                                warn.getErrorCode ());
                        System.out.println ("");
                        warn = warn.getNextWarning ();
                }
        }
        return rc;
}

private static void dispResultSet (ResultSet rs)
        throws SQLException
{
        int i;
        // Get the ResultSetMetaData. This will be used for
        // the column headings
        ResultSetMetaData rsmd = rs.getMetaData ();

        // Get the number of columns in the result set

        int numCols = rsmd.getColumnCount ();
// Display column headings
 //System.out.println("/n numCols:"+numCols);

        for (i=1; i<=numCols; i++) {
                if (i > 1) System.out.print(",");
                System.out.print(rsmd.getColumnLabel(i));
                System.out.print(" \n"+rsmd.getColumnTypeName(i));
        }
        System.out.println("");

        // Display data, fetching until end of the result set

        while (rs.next ()) {

                // Loop through each column, getting the
                // column data and displaying
        for (i=1; i<=numCols; i++) {
                        if (i > 1) System.out.print(",");
                        System.out.print(rs.getObject(i));
                        //System.out.print(rs.getObject(i));
                }
                System.out.println("");

                // Fetch the next result set row

        }
}
private static void tsspServerTest(Connection con)
throws SQLException
{

 ScTsspRecord[] tssp = new ScTsspRecord[2];
 String Brcd="0";
 String Tpd="58177";
 String Acn="100000";
 String Etc="DD";
 String Tamt="1500.00";
 String Trc="40001";
 String[] Crcd = {"USD","1.00000","USD","1"};
 //String[] Crcd = {"USD","1.00000","USD","1","0","125.00","0"};
 String Efd="58177";
 String Vdt="58177";
 String[] Spv = {""};
 String[] Qlf = {""};
 String[] Psi = {""};
```

```
        String[] Msc = {""};
        String Tcmt ="Tax refund";
        String Histn = "";
        String[] Ckreg = {""};
            tssp[0]= new ScTsspRecord (Brcd,
                        Tpd,
                        Acn,
                        Etc,
                        Tamt,
                        Trc,
                        Crcd,
                        Efd,
                        Vdt,
                        Spv,
                        Qlf,
                        Psi,
                        Msc,
                        Tcmt,
                        Histn,
                        Ckreg);

        Brcd="0";
        //Tpd="57446";
        Acn="11110";
        Etc="CI";
        Tamt="1500.00";
        Trc="40002";
        Crcd[0]="USD";
        Crcd[1]="1.000000";
        Crcd[2]="USD";
        Crcd[3]="1";
        //Efd="57446";
        //Vdt="57446";
        //Spv[0]="";
        //Qlf[0]="";
        //Psi[0]="";
        //Msc[0]="";
        Tcmt ="CASH IN";
        //Histn="";
        //Ckreg[0]="";
        tssp[1]= new ScTsspRecord(Brcd,
                                  Tpd,
                                  Acn,
                                  Etc,
                                  Tamt,
                                  Trc,
                                  Crcd,
                                  Efd,
                                  Vdt,
                                  Spv,
                                  Qlf,
                                  Psi,
                                  Msc,
                                  Tcmt,
                                  Histn,
                                  Ckreg);

          CallableStatement cstatmt = con.prepareCall("{call TSSP(?,?)}");
       cstatmt.setObject(1, tssp);
       cstatmt.registerOutParameter(2, -2);
       ResultSet rs = cstatmt.executeQuery();
       dispResultSet(rs);

        }
}
```

# Appendix B. Multiple MTMs Connection Test Code

This appendix provides the lines of code for an example of a TSSP server class called by a Callable Statement.

```
private static String getTsspTest() throws SQLException
{

/****
    {call TSSP(?,?)}
0              !Brcd - Branch Code
58805          !Tpd - Teller Posting Date
39202006       !Acn - Account Number
CD             !Etc - Transaction Code
10000.00       !Tamt - Transaction Amount
102001         !Trc - Transaction Reference Number
USD|1|USD|1    !Crcd* - Currency Code
58796          !Efd - Effective Date
               !Vdt - Value Date
               !Spv* - Supervisor Authorization Information
               !Qlf* - Transaction Qualifiers
               !Psi* - Payment System Instructions
               !Msc* - Miscellaneous Information Code
               !Tcmt - Free Text Transaction Comment
               !Histn - Transaction Notes
               !Ckreg* - Check Register Information
===========================================
0              !Brcd - Branch Code
58805          !Tpd - Teller Posting Date
11110          !Acn - Account Number
CI             !Etc - Transaction Code
10000.00       !Tamt - Transaction Amount
102002         !Trc - Transaction Reference Number
USD|1|USD|1    !Crcd* - Currency Code
58796          !Efd - Effective Date
               !Vdt - Value Date
               !Spv* - Supervisor Authorization Information
               !Qlf* - Transaction Qualifiers
               !Psi* - Payment System Instructions
               !Msc* - Miscellaneous Information Code
               !Tcmt - Free Text Transaction Comment
               !Histn - Transaction Notes
               !Ckreg* - Check Register Information

****/
ScTsspRecord[] tssp = new ScTsspRecord[2];
String Brcd="0";
String Tpd="60229";
String Acn="121";
String Etc="CD";
String Tamt="10000.00";
String Trc="102001";
String[] Crcd = {"USD","1","USD","1"};
String Efd="58796";
String Vdt="";
String[] Spv = {""};
String[] Qlf = {""};
String[] Psi = {""};
String[] Msc = {""};
String Tcmt ="Tax refund";
String Histn = "";
String[] Ckreg = {""};

tssp[0]= new ScTsspRecord (Brcd,
 Tpd,
 Acn,
 Etc,
 Tamt,
 Trc,
 Crcd,
 Efd,
 Vdt,
 Spv,
 Qlf,
 Psi,
 Msc,
 Tcmt,
 Histn,
 Ckreg);

Brcd="0";
Tpd="60229";
Acn="11110";
Etc="CI";
Tamt="10000.00";
Trc="102002";
Crcd[0]="USD";
Crcd[1]="1";
Crcd[2]="USD";
Crcd[3]="1";
Efd="58796";
Vdt="";
Spv[0]="";
Qlf[0]="";
Psi[0]="";
Msc[0]="";
Tcmt ="CASH IN";
Histn="";
Ckreg[0]="";
tssp[1]= new ScTsspRecord(Brcd,
 Tpd,
 Acn,
 Etc,
 Tamt,
 Trc,
 Crcd,
 Efd,
 Vdt,
 Spv,
 Qlf,
 Psi,
 Msc,
 Tcmt,
 Histn,
 Ckreg);

ScProfileForm oProfForm = new ScProfileForm();
String sTSSPMess = oProfForm.getTsspLV(tssp);
System.out.println("getTsspLV output \n"+sTSSPMess);
return sTSSPMess;
            }
```

## Code Example: Using Hashtable for Input Values

```
private static void tsspServerTest(Connection con)

throws SQLException

{

        Hashtable h[] = new Hashtable[2];


        String Acn= "3550330";

        String Brcd= "0";

        String[] Ckreg= {""};
        String[] Crcd= {""};
        String Efd= "";
        String Etc= "CW";
        String Histn= "";
        String[] Msc= {""};
        String[] Psi= {""};
        String[] Qlf= {""};
        String[] Spv= {"*","6399","2153","3550330"};
        String Tamt= "1225.26";
        String Trc= "-346319108";
        String Tcmt= "";
        String Tpd= "";
        String Vdt= "";
        Vector vSpv = new Vector();
        vSpv.add(Spv);

        Hashtable tsspRec1 = new Hashtable();
        tsspRec1.put("BRCD", Brcd);
        tsspRec1.put("TPD", Tpd);
        tsspRec1.put("ACN",Acn);
        tsspRec1.put("ETC",Etc);
        tsspRec1.put("TAMT",Tamt);
        tsspRec1.put("TRC",Trc);
        tsspRec1.put("CRCD", Crcd);
        tsspRec1.put("EFD", Efd);
        tsspRec1.put("VDT", Vdt);
//      tsspRec1.put("SPV", Spv);
        tsspRec1.put("QLF", Qlf);
        tsspRec1.put("PSI",Psi);
        tsspRec1.put("MSC", Msc);
        tsspRec1.put("TCMT", Tcmt);
        tsspRec1.put("HISTN", Histn);
        tsspRec1.put("CKREG", Ckreg);
        tsspRec1.put("VSPV", vSpv);

        Acn= "11110";
        Brcd= "0";
        Ckreg[0]= "";
        Etc= "CO";
        Histn= "";
        Msc[0]= "";
        Psi[0]= "";
        Qlf[0]= "";
        String Spv1[] = {""};
        Tamt= "1225.26";
        Tcmt= "";
        Trc= "-346319106";

        Hashtable tsspRec2 = new Hashtable();
        tsspRec2.put("BRCD", Brcd);
        tsspRec2.put("TPD", Tpd);
        tsspRec2.put("ACN",Acn);
        tsspRec2.put("ETC",Etc);
        tsspRec2.put("TAMT",Tamt);
        tsspRec2.put("TRC",Trc);
        tsspRec2.put("CRCD", Crcd);
        tsspRec2.put("EFD", Efd);
        tsspRec2.put("VDT", Vdt);
        tsspRec2.put("SPV", Spv1);
        tsspRec2.put("QLF", Qlf);
        tsspRec2.put("PSI",Psi);
        tsspRec2.put("MSC", Msc);
        tsspRec2.put("TCMT", Tcmt);
        tsspRec2.put("HISTN", Histn);
        tsspRec2.put("CKREG", Ckreg);

        h[0] = tsspRec1;      // first tssp record
        h[1] = tsspRec2;      // second tssp record

        CallableStatement cstatmt = con.prepareCall("{call TSSP(?,?)}");
        cstatmt.setObject(1, h);
        cstatmt.registerOutParameter(2, -2);
        ResultSet rs = cstatmt.executeQuery();
        dispResultSet(rs);
}
```

# Appendix C. Connection Pooling Parent Class

This appendix provides information on parent classes and how they are used by the Fidelity JDBC Driver in relation to connection pooling. This supplements the information provided in the Connecting Through the Physical Connection Pool section of this document.

- The **ScConnectionPoolDataSource** extends the `sanchez.jdbc.pool.ScDataSource` class and implements `javax.sql.ConnectionPoolDataSource`.
- The **ScDataSource** implements `javax.sql.DataSource, java.io.Serializable`, and `javax.naming.Referenceable`.
- The **ScJDBCConnectionPoolCache** extends the `sanchez.jdbc.pool.ScDataSource` class and implements `java.io.Serializable` and `javax.naming.Referenceable`.
- The **ScJdbcPool** extends the `sanchez.utils.objectpool.ScObjectPool` class.
- The **ScObjectGlobal** extends the `sanchez.base.ScObject` and ScObject implements `java.io.Serializable`.
- The **ScObjectPool** extends `sanchez.base.ScObjectGlobal` and implements `sanchez.utils.objectpool.SiCleanable`.

## Code Example

```
public interface SiCleanable
{
  /**
   * cleanUp()method should invalidate all the objects this
     particular object is referring to.
   * There by making the object garbage collectable.
   */
  public void cleanUp();
   /**
     * This method contains all logic to check consistency
     * of the object state.
      *@see java.lang.Object#finalize()
     */
     public boolean validate();
}
```

# Appendix D.  Data Type Mappings

This appendix shows the mapping between Database data types and JDBC data types.

| Sequence Number | Object Type | Database Data Type |
|---|---|---|
| 1 | BigDecimal | DECIMAL |
| 2 | Boolean | BOOLEAN |
| 3 | Byte | LONG |
| 4 | Short | INTEGER |
| 5 | Integer | INTEGER |
| 6 | Long | LONG |
| 7 | Float | REAL |
| 8 | Double | NUMERIC |
| 9 | Date | DATE |
| 10 | Time | TIME |
| 11 | Timestamp | TIMESTAMP |
| 12 | Blob | BLOB |
| 13 | String | VARCHAR |

# Appendix E. Limitations

Be aware of the following limitations and restrictions in terms of Profile JDBC Driver capabilities.

1. Following are the JDBC 3.0 features that are **not** supported in the Profile JDBC Driver:
   a. Save Points
   b. Passing parameters to `CallableStatement` objects by name
   c. BOOLEAN Data type support
   d. Retrieving and updating the object referenced by Ref object
   e. DataLink/URL data type
   f. Transform groups and type mapping
   g. DatabaseMetadata APIs such as getSuperTables(), getSuperTypes(), and other metadata functions that are used for retrieving database type hierarchies
   h. Multiple ResultSet
2. The following APIs are **not supported** in the Profile JDBC Driver:

   **ResultSet**

```
public Array getArray
public Clob getClob
public Ref getRef
public java.io.InputStream getAsciiStream
public java.io.InputStream getBinaryStream
public java.io.Reader getCharacterStream
public java.io.InputStream getUnicodeStream
public void updateAsciiStream
public void updateCharacterStream
public Object getObject(int i, java.util.Map map)
public boolean rowDeleted()
public boolean rowInserted()
public boolean rowUpdated()
```

   **Statement**

```
public int executeUpdate(String sql, int[] columnIndexes)
public int executeUpdate(String sql, String[] columnNames)
public boolean execute(String sql, int[] columnIndexes)
public boolean execute(String sql, String[] columnNames)
public boolean getMoreResults(int current)
```

   **PreparedStatement**

```
public void setAsciiStream(int parameterIndex, java.io.InputStream x, int readSize)
public void setCharacterStream(int parameterIndex, java.io.Reader reader, int length)
public void setUnicodeStream(int parameterIndex, java.io.InputStream x, int length)
public void setArray(int i, Array x)
public void setRef(int i, Ref x)
public void setURL(int parameterIndex, URL x)
```

**CallableStatement**

```
public Array getArray(int i)
public Clob getClob(int i)
public Ref getRef(int i)
public void setURL(String parameterName, URL val)
```

3. Result Sets - Only Result Sets from MSQL calls can be directly updated. The Result Sets from other server classes such as MRPC, TSSP, HTML, and XML cannot be updated.

4. Unsupported Data Types - The driver supports standard Java SQL types. Fidelity Profile does not support SQL3 data types REF and ARRAY.

5. Update on primary key columns is not supported.

6. Profile does not support NULL values in the database. All numeric columns will store a value of 0 (zero) and text columns will store a space, instead of a NULL value.

7. Database does not allow Float values in exponential format (e.g., 1.4E-45).

8. Data types Real and Float can be used only when CREATE TABLE is used. These data types do not exist in Profile. Tables created using CREATE TABLE are treated as temporary tables in Profile and not all functionalities are available for these tables. It is always advisable to create tables directly in Profile, if possible. The Profile equivalent for Real is Numeric(7,2) and for Float is Numeric(15,2). If the values inserted or updated exceed these lengths, an error will be thrown.

9. When Integer data type columns are created using CREATE TABLE, Profile creates Numeric(5,2) columns. For Integer data types, the decimal precision is 0. A workaround is to create the table with the Numeric(5) data type in Profile.

10. The primary key does not allow decimal precision (number with precision value) in the data definition but it allows storage of decimal values. Hence, the precision length (0) returned may not match with actual data.

11. The Profile database does not support numbers starting with a decimal point (e.g., .999).

12. When creating tables use VARCHAR2(30) for columns that need to store Timestamp data, instead of the Timestamp data type.

13. The following characters are supported in Blob and Memo columns, but are **not supported** in Text columns:

    **\t \n \r null \0**

14. For the Numeric data type in the database, `getObject` would return an object of type Long if the column definition contains no precision. If precision is found, getObject returns an object type of BigDecimal.

15. Scalar functions LENGTH, LOCATE, SQRT, and FULLOUTERJOIN are **not supported** in the Profile database.

16. `getMaxFieldSize` in Statement is not implemented (the database returns a hard-coded value of 1048575 irrespective of what value is set using `setMaxFieldSize`).

17. The `setObject` in PreparedStatement will accept the object only if the object type matches the database type. The object type and database type mappings are listed in **Appendix D**. If the types do not match, an `SQLException` will be thrown with the message "Class Cast Exception. Cannot convert from <Object Type>".

18. All default error messages are available only in English in the `Resource.properties` property file.

# Revision History

This section provides an at-a-glance view of the changes made to this document.

| Rev | Date | Doc Version | Author | Comments |
|---|---|---|---|---|
| 1 | 07/04/05 | 0.1 | L. Chelladurai | Initial draft submitted for Documentation edit |
| 2 | 07/07/05 | 0.2 | H. Warmhold | Includes Documentation edit updates |
| 3 | 07/12/05 | 0.3 | H. Warmhold | Additional Documentation edit updates |
| 4 | 07/19/05 | 0.4 | L. Chelladurai | Included Appendix C |
| 5 | 07/21/05 | 0.5 | H. Warmhold | Includes final Documentation edit updates |
| 6 | 08/03/05 | 0.6 | L. Chelladurai | Updated the technical review feedback given by the SME |
| 7 | 08/05/05 | 1.0 | A. Canfield | Final Doc edit and publication |
| 8 | 06/26/06 | 1.1 | Covansys Team | Updated JDBC 2.0 gap fixes |
| 9 | 06/28/06 | 1.2 | B. Patil | Updated for JDBC 3.0 enhancements |
| 10 | 06/29/06 | 1.3 | M. Thoniyil | Updated documentation/review |
| 11 | 06/30/06 | 1.4 | V. O'Reilly | Reformatted to new template |
| 12 | 07/13/06 | 1.5 | M. Thoniyil | Updated the document with AGGMSG calls |
| 13 | 07/19/06 | 1.6 | H. Warmhold | Documentation edit |
| 14 | 07/19/06 | 1.7 | M. Thoniyil | Edit updates |
| 15 | 08/09/06 | 2.0 | A. Canfield | Final Doc edit and publication |
| 16 | 11/08/06 | 2.1 | M. Thoniyil | Updated connection properties, construction of ScTsspRecord, AGGMSG example, and various appendix examples |
| 17 | 11/14/2006 | 3.0 | M. Brunner | Final Doc edit and publication |