SQL Connection Reference Manual

Version 10.1

SYNERGY DE

Printed: June 2013

DCN SC-01-10.1_02

Synergex
2330 Gold Meadow Way
Gold River, CA 95670 USA

http://www.synergex.com
phone 916.635.7300
fax 916.635.6549

Contents

Preface

About this manual

This manual contains information about Synergy/DE™ SQL Connection. It is written for users who are already familiar with programming and database concepts. It presents some general information on concepts, features, and procedures as they relate to SQL Connection (e.g., the following SQL Server-specific concepts: client-side cursors, dynamic cursors, the firehose cursor, keyset-driven cursors, server-side cursors, and static cursors). This manual includes information on such topics as a starting point, but see your database documentation for complete information.

Manual conventions

Throughout this manual, we use the following conventions:

- In code syntax, text that you type is in `Courier` typeface. Variables that either represent or should be replaced with specific data are in *italic* type.
- Optional arguments are enclosed in *[*italic square brackets*]*. If an argument is omitted and the comma is outside the brackets, a comma must be used as a placeholder, unless the omitted argument is the last argument in a subroutine. If an argument is omitted and the comma is inside the brackets, the comma may also be omitted.
- Arguments that can be repeated one or more times are followed by an ellipsis…
- A vertical bar (|) in syntax means to choose between the arguments on each side of the bar.
- Data types are **boldface**. The data type in parentheses at the end of an argument description (for example, (**n**)) documents how the argument will be treated within the routine. An **a** represents alpha, a **d** represents decimal or implied-decimal, an **i** represents integer, and an **n** represents numeric (which means the type can be **d** or **i**).
- The term "environment variable" refers to logicals on OpenVMS, as well as environment variables on Windows and UNIX platforms.
- To "enter" data means to type it and then press ENTER. ("ENTER" refers to either the ENTER key or the RETURN key, depending on your keyboard.)

- This grid indicates on which platforms and in which environments a routine, statement, etc., is supported: in traditional Synergy™ on Windows (WT), in Synergy .NET on Windows (WN), on UNIX (U), or on OpenVMS (V). By "supported" we mean that the item performs a useful function on that platform or environment. For example, an unsupported routine may cause a compiler error or it may just not do anything.

WT	WN	U	V

WIN

- Items or discussions that pertain only to a specific operating system or environment are called out with the name of the operating system.

Other resources

- ANSI SQL 92 documentation (ANSI X3.135-1992)
- *xfODBC User's Guide*
- *Synergy DBL Language Reference Manual*
- *Environment Variables & System Options*
- *Installation Configuration Guide*
- Connectivity Series release notes (**REL_CONN.TXT**)

Product support information

If you cannot find the information you need in this manual or in the resources listed above, you can reach the Synergy/DE Developer Support department at the following numbers:

800.366.3472 (in the U.S. and Canada)
916.635.7300 (in all other locations)

To learn about your Developer Support options, contact your Synergy/DE account manager at one of the above numbers.

Before you contact us, make sure you have the following information:

- The version of the Synergy/DE product(s) you are running
- The name and version of the operating system you are running
- The name and version of the database system you are running
- The hardware platform you are using
- The error mnemonic and any associated error text (if you need help with a Synergy/DE error)
- The statement at which the error occurred

- The exact steps that preceded the problem
- What changed (for example, code, data, or hardware) before this problem occurred
- Whether the problem happens every time and whether it is reproducible in a small test program
- Whether your program terminates with a traceback, or whether you are trapping and interpreting the error
- Returned error information from %SSC_GETEMSG

Reporting Synergy .NET issues

If you are having any of the following problems, please send us the complete set of source files to re-create the issue, and send us the information in the **BuildVersion.txt** file, which is in MSBuild\Synergex\VS2010 under the Program Files directory or the Program Files (x86) directory.

- Visual Studio lock up or crash
- Compiler crash
- Unusual MSIL Assembler (**ilasm.exe**) issues
- "Invalid program" errors
- "JIT Compiler has encountered an internal limitation" error at runtime

For Visual Studio issues, zip the entire project.

Note that for untrapped errors, you won't get a traceback, as you would with traditional Synergy. Instead, you'll get the Windows Dr. Watson box. And if you click Debug, you'll go into the debugger. If the program was not built with debug information, and you instead click Cancel, you'll get a traceback.

Synergex Professional Services Group

If you would like assistance implementing new technology or would like to bring in additional experienced resources to complete a project or customize a solution, Synergex® Professional Services Group (PSG) can help. PSG provides comprehensive technical training and consulting services to help you take advantage of Synergex's current and emerging technologies. For information and pricing, contact your Synergy/DE account manager at 800.366.3472 (in the U.S. and Canada) or 916.635.7300.

Comments and suggestions

We welcome your comments and suggestions for improving this manual. Send your comments, suggestions, and queries, as well as any errors or omissions you've discovered, to **doc@synergex.com**.

1

Welcome to SQL Connection

What Is SQL Connection?

SQL Connection is an application program interface (API) that enables Synergy applications to use SQL-based functions to access and manipulate data from various database systems. It consists of two types of functions that are based on standard SQL-based operations:

- The *database functions* are directly related to SQL-based operations and data access. They greatly simplify application development by reducing the total number of calls needed to accomplish a wide variety of SQL functions. See chapter 3, "Database Functions."
- The *utility functions* enable you to get information, map error codes, and set date and time options during the execution of your Synergy application. See chapter 4, "Utility Functions."

Each SQL Connection function generates a value and can be used any place a literal can be used in a Synergy program. Except where noted, SQL Connection functions work with both traditional Synergy and Synergy .NET.

There is a glossary at the end of this manual to clarify some of the terms we use, but throughout the manual we assume you are familiar with relational database management system (RDBMS) and SQL concepts. For information on a specific database operation, you may need to refer to the documentation for your database.

Features and supported databases

SQL Connection conforms to ANSI-standard database communication methods SQLCA and SQLDA (ANSI 89) and supports

- MySQL (version 5.1 and higher) on supported Windows, Linux, and AIX platforms.
- Oracle® (version 10 and higher) on all supported Windows, UNIX, and OpenVMS platforms. Use version 10.2.0.4 or higher for OCI-related fixes.
- SQL Server (version 2008 and higher) on supported Windows platforms. Note that for SQL Server 2008, only the VTX12_SQLNATIVE driver is supported.
- Synergy DBMS (version 7.1 and higher) on all supported Windows, UNIX, and OpenVMS platforms.

SQL Connection has limited support for the following database systems. Note that support for these databases may require assistance from Synergex Professional Services and additional support fees. Contact your Synergy/DE account manager for details. (See "Product support information" on page viii.)

- Informix® on UNIX systems
- ODBC-compliant databases on Windows systems
- Oracle Rdb on OpenVMS systems
- Sybase® on Windows and UNIX systems

Additional SQL Connection features include

- record-oriented or column-oriented bulk data access.
- built-in statement caching and SQL prefetch caching.
- automatic data-type conversion and binding of Synergy DBL variables with data.
- independence from Synergy DBL file I/O operations.
- support for up to seven concurrent database connections per program on UNIX and OpenVMS, and up to 100 concurrent connections per program on Windows (for multi-threading). You can open multiple databases simultaneously, and you can open the same database more than once in an application (under the same user name or under a different user name).

Components and configurations

SQL Connection uses database drivers as the liaison between third-party data sources and your Synergy application. These drivers enable Synergy applications to use database-independent, SQL-based queries to access data from a variety of local or remote RDBMSs. For the most part, these drivers reside on the same system as the data source, and each driver makes calls that conform to a programming interface for the database (e.g., OCI for Oracle). See figure 1-1 below.

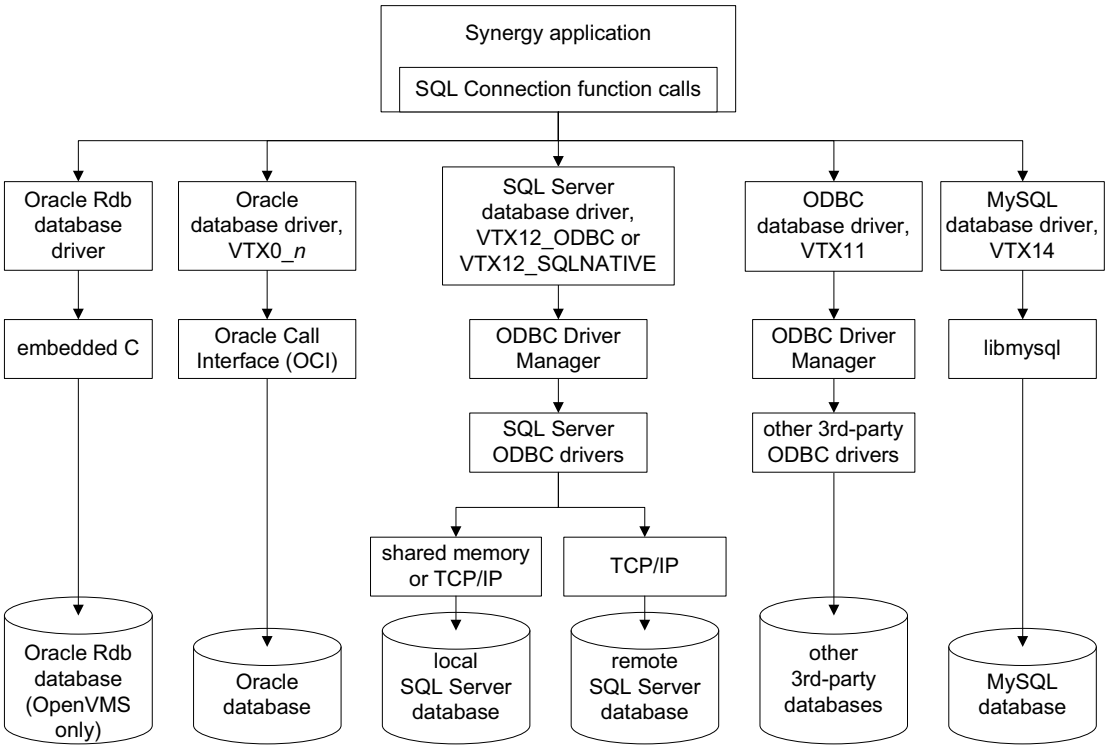


Figure 1-1. Database drivers and interfaces.

For client/server configurations, there are two ways to access remote data. You can use the network component for Connectivity Series, SQL OpenNet, for the network layer, or you can connect directly to a local client for the database and rely on the database to provide the network layer. We recommend using SQL OpenNet. If you connect directly to the database client, the database driver must be on each client. However, if you use SQL OpenNet, you need to put the database driver only on the server, not the clients. (The only driver that clients need is VTX3, which is the SQL OpenNet client. See figure 1-2 on page 1-5.) The primary advantage to using SQL OpenNet, however, is that you'll generally get better performance.

For a stand-alone configuration, you can generally connect directly using the Synergy database driver for the database.

For information on connecting to databases, see "Building Connect Strings" on page 2-16.

For information on configuring SQL Connection and SQL OpenNet, see the "Configuring Connectivity Series" chapter of the *Installation Configuration Guide*.

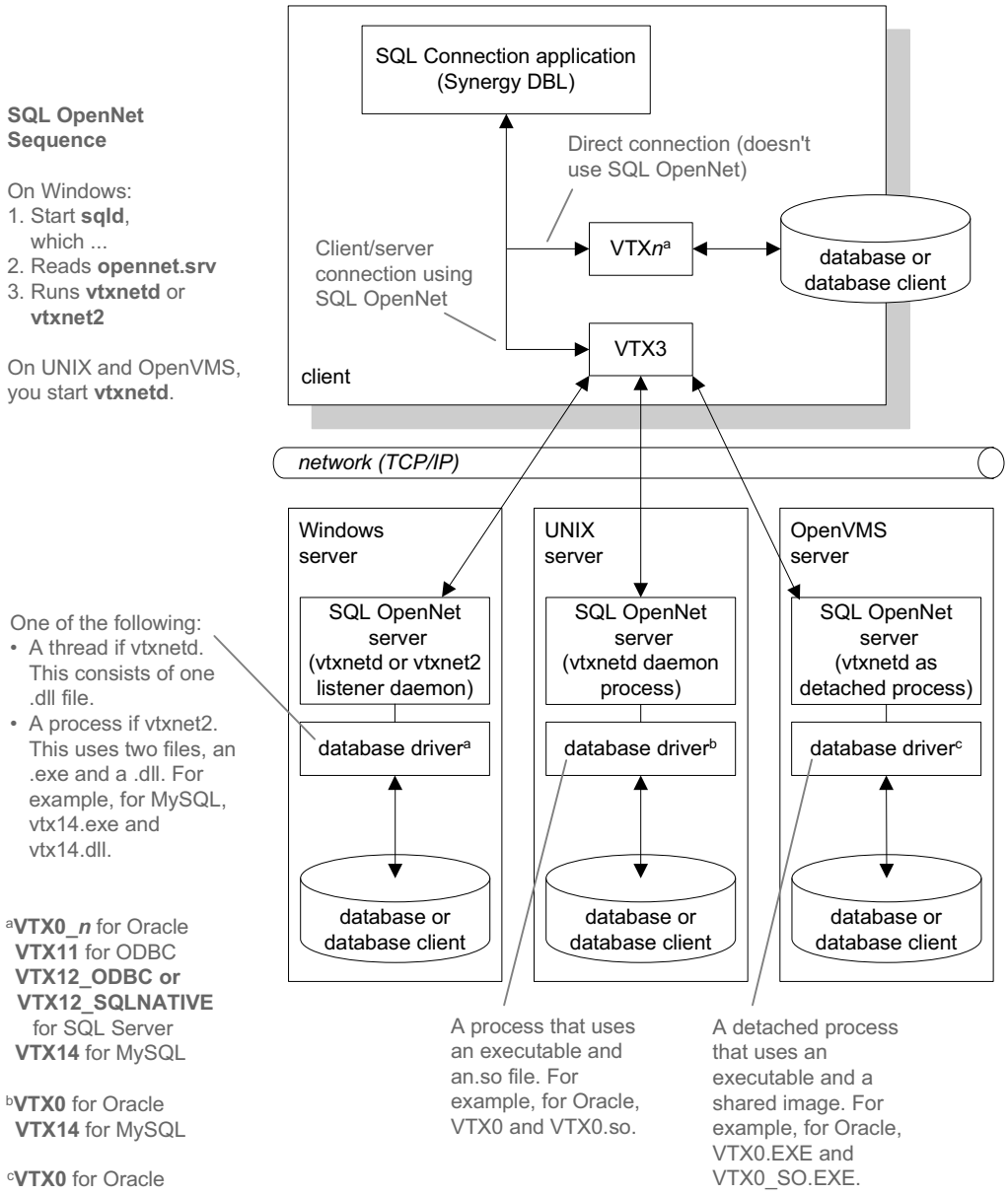


Figure 1-2. SQL OpenNet connections and direct connections (client/server and stand-alone).

Installing, Configuring, and Initializing

To run an application that uses the SQL Connection API,

1. Install and configure SQL Connection and SQL OpenNet and start the SQL OpenNet server by following the steps in the installation instructions and the "Configuring Connectivity Series" chapter of the *Installation Configuration Guide*. The SQL OpenNet service has a display name of "Synergy/DE OpenNet Server" and a service name of **SynSQL**.
2. Initialize SQL Connection. This instructs the Synergy runtime to allocate the necessary memory.

   On OpenVMS, call %INIT_SSQL from your program. For information, see "Initializing SQL Connection" on page 2-5.

   On Windows and UNIX, there are two ways to do this:

   - Call %OPTION from your program and use this function to set system option 48. For information, see "Initializing SQL Connection" on page 2-5.
   - Use the DBLOPT environment variable to set system option 48. You can set DBLOPT in the system environment or (on Windows only) in **synergy.ini**. For information on DBLOPT, see DBLOPT in the "Environment Variables" chapter of *Environment Variables & System Options*.

   If you use DBLOPT to set system option 48, all Synergy programs use the extra memory that Synergy runtime allocates for SQL Connection, even if these programs don't use SQL Connection. However, if you call %INIT_SSQL or %OPTION from your program, only programs that use SQL Connection will use these extra resources.

Setting Options and Environment Variables

This section discusses the options you can set for SQL OpenNet. For information on

- configuring SQL Connection (and SQL OpenNet) for stand-alone or client/server access, see the "Configuring Connectivity Series" chapter of the *Installation Configuration Guide*.
- database-specific options, see %SSC_CMD on page 3-7.
- date and time options, see %SSC_OPTION on page 4-8.
- logging options, see "Troubleshooting and Error Logging" on page 5-2.
- VORTEX_HOST_HIDEGPF, an environment variable that prevents the SQL OpenNet server from shutting down if a thread fails, see VORTEX_HOST_HIDEGPF in the "Environment Variables" chapter of *Environment Variables & System Options*.
- VORTEX_HOST_NOSEM, an environment variable that causes SQL OpenNet to crash when there's an access violation (enabling you to attach to the Windows debugger), see VORTEX_HOST_NOSEM in the "Environment Variables" chapter of *Environment Variables & System Options* for more information. (Note that this should only be used at the request of Synergy/DE Developer Support.)

For information on *how* to set environment variables, see "Setting Environment Variables and Initialization Settings" in the "Environment Variables" chapter of *Environment Variables & System Options*.

Using network initialization files to set network defaults

There are three initialization files you can use to set network defaults:

- **Net.ini** enables you to specify an encryption key for the client and other options.
- **Opennet.srv** enables you to specify the filenames and parameters for the services run by the Synergy/DE OpenNet Server service (**SynSQL**) on Windows.
- **Startnet** (UNIX) and **STARTNET.COM** (OpenVMS) enable you to set environment variables on the server. (These also kill and restart SQL OpenNet server.)

Setting connect string defaults and encryption in net.ini

The **net.ini** file enables you to specify an encryption key for the client, specify time-outs, instruct SQL Connection functions to return error codes for communication errors, and other options. When you install Connectivity Series, the installation creates a default **net.ini** file with a default encryption setting. To change settings in **net.ini**, use a text editor. Note the following:

- We recommend that you don't change any **net.ini** setting except key_connect.
- The **net.ini** file must be on the client in the connect\synodbc\lib directory.

- The Connectivity Series installation (Windows), **setsde** (UNIX), or SYS$MANAGER:CONNECT_STARTUP.COM (OpenVMS) sets the VORTEX_HOME environment variable to the connect\synodbc directory, which contains the lib directory. Do not change this setting.

  Note that because VORTEX_HOME is set at the system level, if you install both 64-bit and 32-bit versions of Connectivity Series on the same 64-bit Windows machine, the last version installed determines the VORTEX_HOME setting by overwriting the previous setting. So if you do change **net.ini**, you'll need to make sure you change the correct one.

- There can't be any control characters in the **net.ini** file. If there are, you'll get an "invalid integer" error when you connect to the database.
- The **net.ini** file is not overwritten when you upgrade Connectivity Series, nor is it removed when you uninstall. We distribute a file named **net_base.ini** (also located in the connect\synodbc\lib directory), which contains default settings and can be used as a reference.

The following options can be set in **net.ini**.

key_connect — Specifies a key for the algorithm used to encrypt user names and passwords for the database and for the host (if **vtxnetd** or **vtxnet2** is also started with the **-k** option). This encrypts user names and passwords being sent across the wire. Use the following syntax:

```
key_connect n
```

where *n* is any number between 1 and 2,147,483,647. Note that *n* must be set to the same value on both the client (where it is set with this **net.ini** option) and the server (where it is set with the **-k** option on the **vtxnetd** or **vtxnet2** command line). See "The vtxnetd and vtxnet2 Programs" in the "Configuring Connectivity Series" chapter of the *Installation Configuration Guide*.

packetsize — Sets the minimum network packet size used by SQL OpenNet. The default is 8192 bytes. This option defines a minimum size for an aggregate buffer, which is a buffer created when data for multiple network packets needs to be sent to the client. This mechanism reduces network traffic by combining packets and sending them as a unit with the specified minimum size. To set this, use the following syntax, where *size* is the size in bytes:

```
packetsize size
```

Note that changing the default packet size *may cause performance problems*. This setting should not be less than the size of the prefetch buffer (specified with the *bufsize* argument for %SSC_INIT) plus 200 bytes. If you are using a WAN (wide area network), you may want to change this value to reduce load on the network. The packet size used by the SQL OpenNet server is set by the packetsize setting in the **net.ini** file on the client.

port — Sets the communication port number (which defaults to the vtxnet setting in the **services** file). Use the following syntax:

```
port port_number
```

See "Specifying the port number" in the Windows section of the "Configuring Connectivity Series" chapter of the *Installation Configuration Guide*.

read_timeout — Specifies how long (in seconds) SQL OpenNet should wait for a read operation to complete. Use the following syntax:

```
read_timeout time
```

By default this is set to 0, which prevents a time-out.

return_errno — Instructs SQL OpenNet to return an operating system error code (rather than -1) if there's a communication error. For example, if an attempt to connect to a database on a server generates the TCP/IP socket error 10061, the %SSC_CONNECT function will return "10061" if you set this option to yes, but it will return "-1" if this option is set to no). Use the following syntax:

```
return_errno yes|no
```

By default return_errno is set to no.

write_timeout — Specifies how long (in seconds) SQL OpenNet should wait for a write operation to complete. Use the following syntax:

```
write_timeout time
```

By default this is set to 0, which prevents a time-out.

The following example **net.ini** file sets the service to connect to a Windows machine named **mywinserver**, instructs the server to use the Oracle driver, sets the encryption key to 6541, and so forth.

```
rem        SQL OpenNet init file
hostnamesvc0 mywinserver!vtx0_10
key_connect 6541
packetsize 1300
port 1990
return_errno yes
read_timeout 60
write_timeout 60
hostenv0 ENV1=env_var_spec,ENV2=env_var_spec2
```

Using opennet.srv to set SQL OpenNet services and environment variables

To specify filenames and parameters for services run by SQL OpenNet on a Windows system, add the service as a line in the **opennet.srv** file. Use the following syntax:

srv_file `[-p`*port_no*`] [log]`

where *srv_file* is the path and filename for the service (either **vtxnet2** or **vtxnetd**) and *port_no* is the port number. The log option creates a log file with a **.log** extension in the directory that contains the service. If *port_no* is not specified, the default port number, 1958, is used.

For example, the following line executes the vtxnet2 service on port 1660 with logging:

```
vtxnet2.exe -k67834 -p1660 log
```

To set an environment variable in **opennet.srv**, use the following syntax:

env_variable=setting

The environment variable setting should precede the line that starts the service. For example:

```
VORTEX_HOST_LOGFILE=c:\vortex
vtxnet2.exe -k67834 -p1660 log
```

For more information, see "Customizing the opennet.srv file" in the Windows section of the "Configuring Connectivity Series" chapter of the *Installation Configuration Guide*.

Using startnet to set environment variables

The **startnet** (UNIX) and **STARTNET.COM** (OpenVMS) files start SQL OpenNet Server. You can also set environment variables in these files. Environment variables set in these files will be set on the server when **startnet** or **STARTNET.COM** are run. Use the following syntax:

- In **startnet** (UNIX): *env_variable=setting*; export *env_variable*
- In **STARTNET.COM** (OpenVMS): define *env_variable setting*

Environment variable settings should precede the line that starts the SQL OpenNet server. For example:

```
VORTEX_HOST_LOGFILE=usr2/vortex; export VORTEX_HOST_LOGFILE
nohup vtxnetd -p1958 log &
sleep 1
```

2

Creating SQL Connection Programs

Writing an SQL Connection Program

The Connectivity Series distribution includes sample programs that illustrate the basic structure for an SQL Connection program. These programs are in the connect\synsqlx subdirectory of the main Synergy/DE installation directory.

The following examples work with MySQL, Oracle, SQL Server, Sybase, and Synergy databases. They are distributed on all platforms.

- **exam_create_table.dbl** — Creates a table and inserts some rows of data. Run this program first. Note that if you uncomment the ".define MULTI_ROW_INSERT" line, this program performs a bulk insert.
- **exam_fetch.dbl** — Contains a simple query. Retrieves rows inserted by **exam_create_table.dbl**.
- **exam_multirow_fetch.dbl** — Contains a multi-row query. Retrieves rows inserted by **exam_create_table.dbl**.
- **exam_fetch_update.dbl** — Contains a simple update. Retrieves rows inserted by **exam_create_table.dbl** (by using a bind variable in the WHERE clause) and then updates the data using %SSC_SQLLINK. Uses a cached cursor to update additional data by calling %SSC_REBIND with a different value for the bind variable.

The following example is for MySQL only. This is distributed only on Windows and UNIX platforms.

- **stp_mysql.dbl** — Calls a stored procedure to access table created by **exam_create_table.dbl**.

The following example is for Oracle only. This is distributed on all platforms.

- **stp_ora.dbl** — Calls a stored procedure to access table created by **exam_create_table.dbl**.

The following examples are for SQL Server only. These are distributed only on Windows platforms and will run only if the Microsoft-supplied pubs sample database is installed. (See the comments in the files for details.)

- **stp_odbc.dbl, stp_sqlsrv1.dbl, stp_sqlsrv2.dbl** — Call stored procedures to access the table created by **exam_create_table.dbl**. These all work with VTX12_ODBC and VTX12_SQLNATIVE.

  (Connectivity Series also includes **stp_sqlsrv.dbl**, which works with the deprecated VTX12_2000 database driver.)

The Connectivity Series distribution (all platforms) also includes **exam_saveviews.dbl**, an example program for saving views for *xf*ODBC.

VMS

When linking SQL Connection programs on OpenVMS, be sure to use the **ssqlrtl.opt** link options file (instead of **synrtl.opt**).

Basic program structure

To ensure your program works properly with the SQL Connection API, write your program to do the following:

- Include the **ssql.def** file. (See "Including ssql.def" on page 2-4.)
- Allocate a data area. (See "Allocating a data area" on page 2-4.)
- Initialize SQL Connection. (See "Initializing SQL Connection" on page 2-5.)
- Connect to a database. (See "Database connections" on page 2-5.)
- Open one or more cursors. (See "Cursors" on page 2-27.)
- Define variables and bind data areas. (See "Data Mapping" on page 2-33.)
- Process the required SQL statements for each cursor. (See "Processing SQL statements" on page 2-7.)
- Close the cursors. (See "Cursors" on page 2-27.)
- Disconnect from the database. (See "Database connections" on page 2-5.)

Note that the sequence of the above and the details vary. For example, SELECT statements are typically processed as follows:

1. Assign values to bind area.
2. Open a cursor and bind data variables.
3. Define data for the result row.
4. Fetch and load data into defined data fields.
5. Process the fetched data.
6. Repeat the fetch, load, and process steps until complete.
7. Close the cursor.

Non-SELECT statements, on the other hand, are typically processed as follows:

1. Open a cursor and bind data variables.
2. Assign values to the bind area.
3. Execute the statement. (This does the actual bind.)
4. Repeat the assign and execute statements until complete.
5. Close the cursor.

See "Function Call Flow" on page 2-9 for details on the function call sequences used for different types of queries and updates.

Including ssql.def

The **ssql.def** file contains the definitions needed for SQL Connection. You must use the .include compiler directive in your SQL Connection program to include **ssql.def**, which is located in the synergyde/connect directory. Because the CONNECTDIR environment variable is set generally set to this directory, you can use CONNECTDIR in the .include statement to locate **ssql.def**. For example:

```
.include "CONNECTDIR:ssql.def"
```

For more information, see .INCLUDE in the "Preprocessor and Compiler Directives" chapter of the *Synergy DBL Language Reference Manual*. And for more information on CONNECTDIR, see "Appendix A: Environment Variables" in the *xfODBC User's Guide*.

Allocating a data area

To receive data from or send data to a database, your program must allocate a data area for the incoming and outgoing data. To enable your program to do this, create a record with variables that correspond to the data you'll be reading from or writing to the database. For example:

```
.align
literal samples     ;Data area for retrieved data (Variables for defining)
        deptnum         ,i4
        deptname        ,a6
        division        ,a15

.align
record data          ;Data area for data to insert (Variables for binding)
       s_deptnum       ,[MX_VARS] i2    ,1,2,3,
       s_deptname      ,[MX_VARS] a6    ,"SDM","SUPP","ACCT"
       s_division      ,[MX_VARS] a15   ,"ACCOUNT", "INTERNAL S"
&                                       ,"OFFICE MIS"
```

For more information on data area allocation, see the "Defining Data" chapter of the *Synergy DBL Language Reference Manual*.

Initializing SQL Connection

For an SQL Connection program to run, you must initialize SQL Connection. This instructs the Synergy runtime to allocate the necessary memory.

You can initialize SQL Connection with the DBLOPT environment variable (see "Installing, Configuring, and Initializing" on page 1-6), or you can initialize SQL Connection from your program. To initialize SQL Connection from your program, do one of the following:

- For Windows and UNIX systems, use the %OPTION function to set system option 48. For example:

  ```
  sts = %option(48,1)    ;Sets option 48 and assigns the return value to
                         ; a variable (sts).
  ```

  For more information, see %OPTION in the "System-Supplied Subroutines and Functions" chapter of the *Synergy DBL Language Reference Manual*.

- For OpenVMS systems, call the %INIT_SSQL function. For example:

  ```
  xcall init_ssql        ;Initializes SQL Connection.
  ```

  For more information, see %INIT_SSQL on page 3-2.

Database connections

Connecting to a database essentially means logging on to a database. Once logged on, a connection is established and maintained by SQL Connection as an open database channel. Once a connection is established, each SQL statement is processed by the native SQL processor within the database engine. SQL Connection simply passes the SQL statement through to the database.

Be careful not to exceed the maximum number of concurrent connections. On UNIX and OpenVMS, SQL Connection enables you to maintain up to seven concurrent connections to one or more heterogeneous databases in a client/server configuration. On Windows, to accommodate multi-threading, you can have up to 100 connections. If your application deals with multiple database servers, you may want to connect to the same database more than once, but remember that each connection requires a separate log-on to the database. On the other hand, you can disconnect and reconnect to the same database without increasing the number of connections.

.NET

Note that with Synergy .NET in a multi-threaded environment, you must ensure that each thread uses a separate connection (i.e., a separate channel), and note that channels are not automatically shut down when threads terminate.

We don't recommend recycling (pooling) connections, particularly for Oracle databases. You may want to consider recycling connections if there is generally a large period of elapsed time (e.g., over 15 minutes) between connection attempts, but you should never recycle Oracle connections because the OCI library that SQL Connection uses can't handle connection recycling well.

Connecting to a database

To connect to a database, use the %SSC_INIT function to initialize a database channel, and then use the %SSC_CONNECT function with a connect string to connect to a database channel (and specify the database, user name, password, and so forth). The syntax of the connect string is dependent on the database and configuration. (See "Building Connect Strings" on page 2-16.)

```
if (%ssc_init(dbchn))              ;Initialize a database channel.
  goto err_exit

Writes(1, "Connecting to database. Please wait...")
if (%ssc_connect(dbchn, user))     ;Connect to database using the connect
  goto err_exit                    ; string represented by the user
                                   ; variable and the database channel
                                   ; represented by dbchn.
```

For more information, see %SSC_INIT on page 3-35 and %SSC_CONNECT on page 3-19.

Disconnecting from a database

To disconnect from a database and release the database channel, use the %SSC_RELEASE function. For example:

```
if (%ssc_release(dbchn)) ;Release the database channel
  goto err_exit          ; represented by the dbchn variable.
```

For more information, see %SSC_RELEASE on page 3-50.

Using cursors

SQL Connection uses two types of cursor: logical and database. When you initialize a database channel with %SSC_INIT, you'll allocate logical cursors, but otherwise, you'll create, use, and close database cursors. %SSC_OPEN opens a new cursor, %SSC_SCLOSE and %SSC_CLOSE close cursors (soft close or hard close), and various SQL Connection routines accept a cursor ID so they can access data and update the database via a cursor. You can also use %SSC_CMD to set cursor options. See "Cursors" on page 2-27 for information.

Defining variables and binding data areas

You can use variables to store data sent to and received from the database. Defined variables store data received from a database; bind variables store data that's sent to a database. To create defined variables, you use %SSC_DEFINE or %SSC_STRDEF. For a bind variable, you pass a placeholder in the SQL statement and then pass the bind variable in %SSC_OPEN, %SSC_BIND, or %SSC_SQLLINK. See "Defining variables and binding data areas" on page 2-6 for information.

Processing SQL statements

Once you open a cursor and, if necessary, map the data, you can process an SQL statement in one of three ways, depending on type of SQL statement. See table below.

Type	Function	Description
SELECT statement	%SSC_MOVE	Loads desired data to the program data area.
Non-SELECT statement	%SSC_EXECUTE	Executes an SQL statement.
Stored procedure	%SSC_EXECIO	Executes a stored procedure and passes parameters, but doesn't accept a result set.
	%SSC_EXECUTE	Executes a stored procedure without parameters or a result set.
	%SSC_MOVE	Retrieves data from a SQL Server stored procedure result set.

Supported SQL statement syntax differs from one database to another. However, most databases support the ANSI SQL syntax. If you want to write a truly database-independent application, you must avoid database-specific syntax in your SQL statements.

Stored procedures are highly recommended to boost database performance. Stored procedures can enhance program modularity and reduce development costs. In a client/server configuration, well-written stored procedures can also greatly reduce network traffic. Note, however, that stored procedures are database dependent and therefore cannot be used in a truly database-independent application. See "Stored Procedures" on page 2-51 for more information.

Using your program with different drivers and databases

Keep the following in mind when working with different databases and drivers:

- Connect string syntax and connection issues differ for each database driver. See "Building Connect Strings" on page 2-16 for information.
- Some databases have their own SQL constructs which won't work with other databases. Avoid database-specific SQL if you plan to use your program with different databases.
- Stored procedure syntax differs for each database. For example, use the ODBC CALL escape sequence if you are using an ODBC driver:

  ```
  sqlp = "{call read_salary (:1, :2, :3)}"
  if (%ssc_open(dbchn, cur1, sqlp, SSQL_NONSEL))
  ```

  For more information, see "Stored Procedures" on page 2-51 and the **stp_*.dbl** example files included in the Connectivity Series distribution.

- Different databases and database drivers produce different errors. If your program was written to handle errors from one database and driver, don't expect to get the same errors when you move to another database or use another database driver. For example, if you use an ODBC database driver, you will get ODBC errors.
- Different drivers use different %SSC_CMD options. See %SSC_CMD on page 3-7.
- Cursors work differently for different databases. See "Cursors" on page 2-27.
- Autocommit works differently on different databases, so transactions that make sense for one database/driver combination may not work for another. For example, if you accessed SQL Server with the deprecated VTX12_2000 database driver, any operation that modified the database (UPDATE, DROP TABLE, etc.) would have triggered an implicit commit unless there was an explicit transaction in progress, so there was no need to call to %SSC_COMMIT after one of these operations. On the other hand, with Oracle databases, databases accessed with VTX11, or SQL Server databases when accessed with VTX12_SQLNATIVE, autocommit is turned off by default, so you *do* need to use %SSC_COMMIT to commit the transaction before calling %SSC_RELEASE. If you don't, you will lose changes; %SSC_RELEASE rolls back pending transactions. For more information, see "Transactions and Autocommit" on page 2-49.
- Updates and locking work differently for different databases. See "Updates and Locking" on page 2-42.

Note the following for SQL Server, and for more notes and tips on using SQL Server with SQL Connection, see "SQL Server notes and examples" on page 2-23.

- We recommend that you use **vtxnetd**, rather than **vtxnet2**.
- Unlike MySQL, Oracle, and Synergy databases, all columns and indices for SQL Server databases default to a case-insensitive ordering using the SQL_Latin1_General_CP1_CI_AS collation sequences. This means that reports using an ORDER BY, an index, or a relational operator (>, <, ==, and so forth) in an SQL statement will work differently for SQL Server than for other databases (including Synergy databases, which use an ANSI collation sequence). Additionally, for characters such as underscore (_) and dash (-), SQL Server will return results in an order that's different than other databases, even though for both ASCII and Synergy DBL, an underscore is considered higher than a dash.
- For information on connecting to SQL Server on Windows clusters, see the Synergex KnowledgeBase article 100000654.

Function Call Flow

The following diagrams illustrate the function call flow for each of the primary SQL commands that you might use in your SQL Connection program:

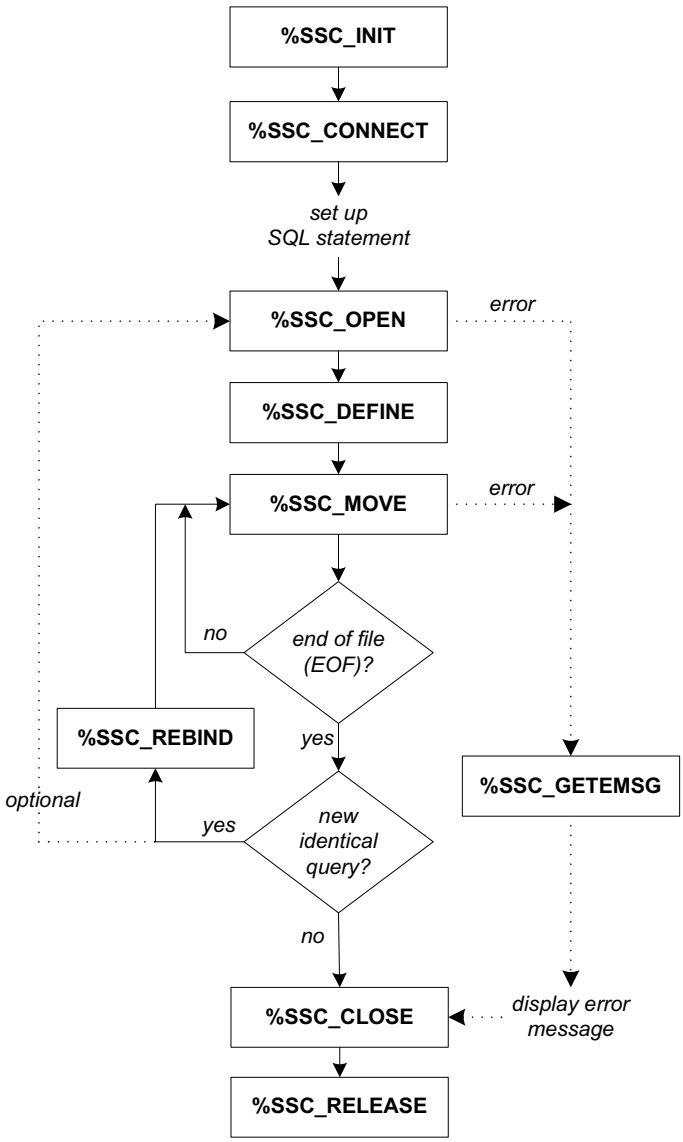
Simple query

1. Initialize SQL Connection.
2. Connect to database.
3. Set up SQL statement.
4. Open cursor with SSQL_SELECT and SSQL_STANDARD.
5. Test for errors.
6. Define destination variables.
7. While rows are available, fetch data with %SSC_MOVE.
8. At end of query, if new identical query, use %SSC_OPEN or %SSC_REBIND.
9. Close cursor.
10. Release connection if no more operations.

    For examples, see

    - **exam_fetch.dbl**
    - **exam_fetch_update.dbl** (for %SSC_REBIND example)

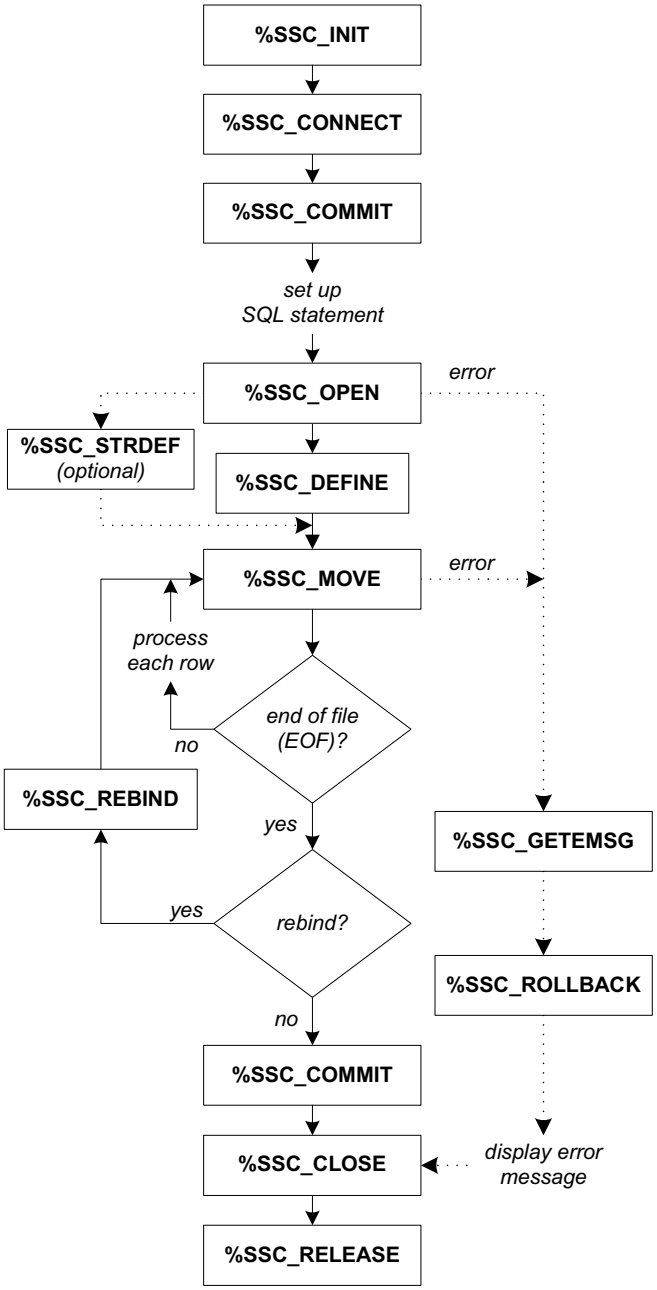
    See "Writing an SQL Connection Program" on page 2-2 for information on these programs.



Multirow query

1. Initialize SQL Connection.
2. Connect to database.
3. Set up SQL statement.
4. Open cursor with SSQL_SELECT and SSQL_STANDARD.
5. Test for errors.
6. Define destination variables or use arrays for %SSC_DEFINE.
7. Fetch data with %SSC_MOVE.
8. If not end of file (EOF), process returned data for each returned row and get more data. (The number of rows returned is available as the *row_count* argument for %SSC_MOVE.)
9. Rebind if required.
10. End transaction by committing it or by rolling it back if there's an error.
11. Release connection if no more operations.

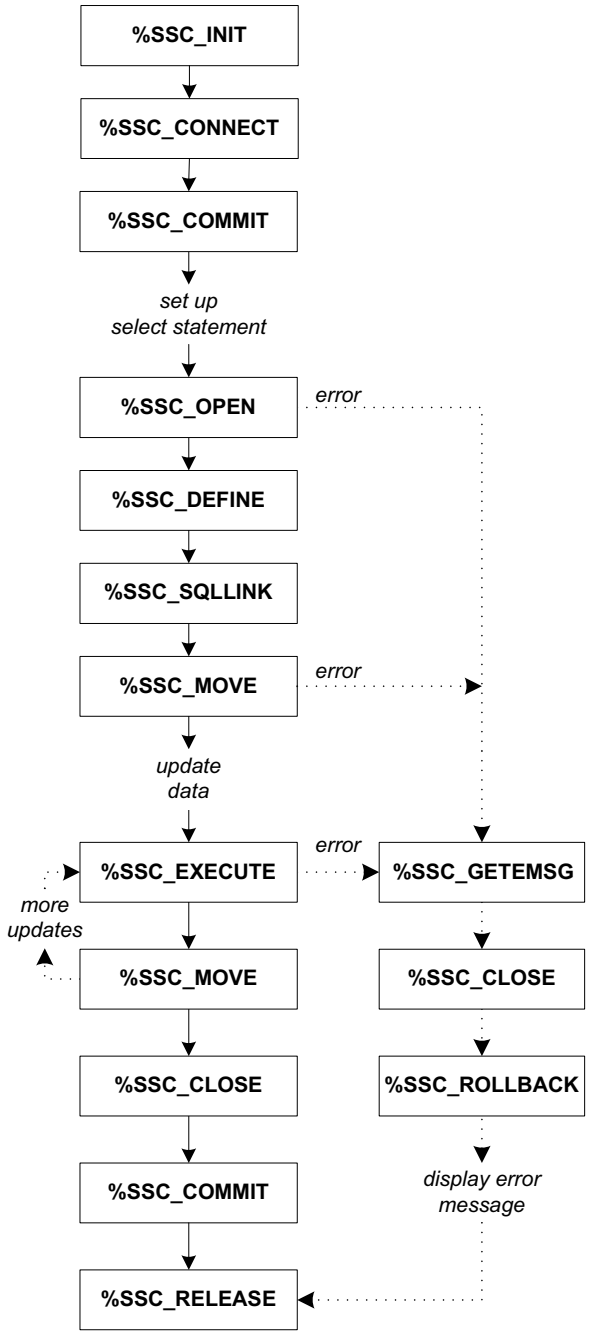
    For an example, see **exam_multirow_fetch.dbl**. See "Writing an SQL Connection Program" on page 2-2 for information on this program.



Simple atomic update

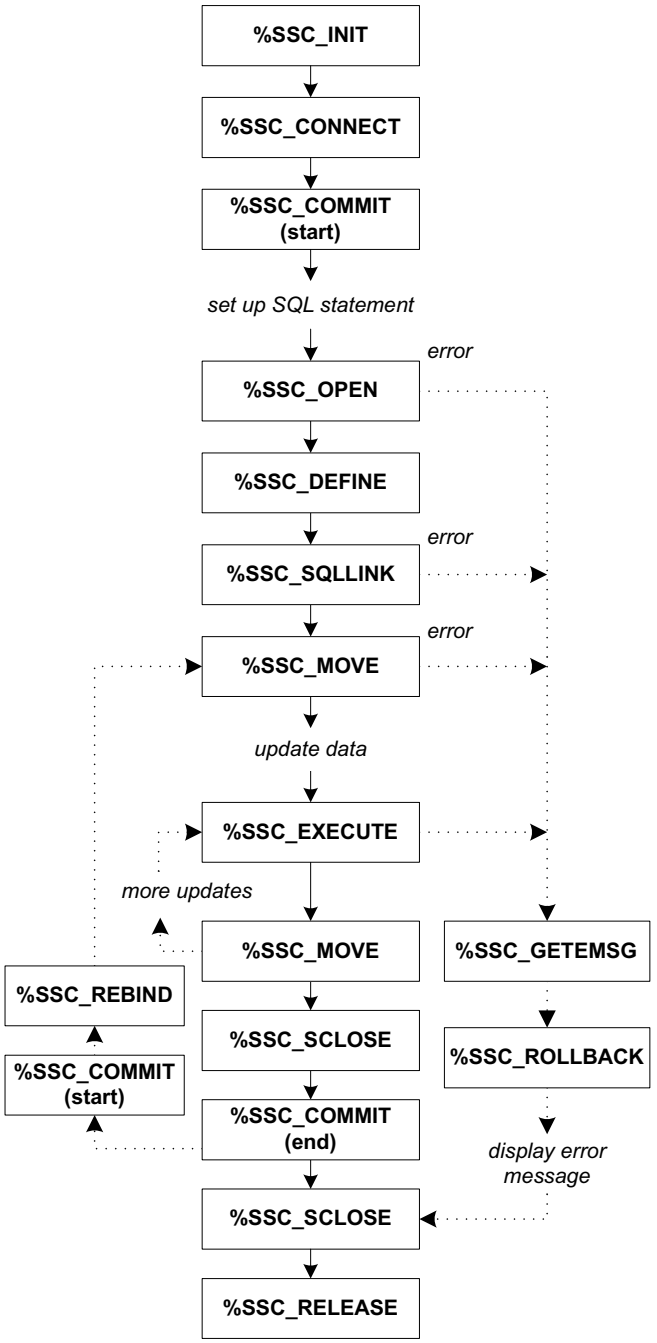
1. Initialize SQL Connection.
2. Connect to database.
3. Start transaction with %SSC_COMMIT.
4. Set up SELECT statement.
5. Open cursor with SSQL_SELECT and SSQL_POSITION, binding variables as necessary.
6. Define variables.
7. Set up update statement with %SSC_SQLLINK, binding variables for update.
8. Fetch data with %SSC_MOVE.
9. Execute update with SSQL_STANDARD.
10. Close cursor.
11. End transaction by committing it or by rolling it back if there's an error.
12. Release connection if no more operations.

    For an example, see **exam_fetch_update.dbl**. See "Writing an SQL Connection Program" on page 2-2 for information on this program.



Bulk update

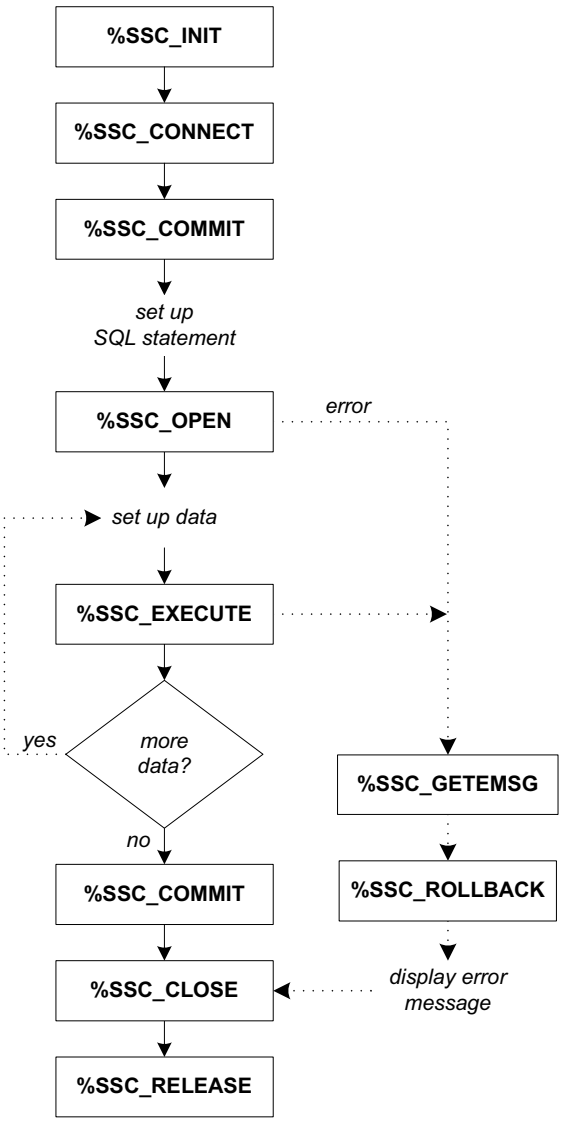
1. Initialize SQL Connection.
2. Connect to database.
3. Start transaction with %SSC_COMMIT.
4. Set up SQL statement.
5. Open cursor with SSQL_SELECT and SSQL_POSITION, binding variables as necessary.
6. Define variables.
7. Set up update statement with %SSC_SQLLINK, binding variables as necessary.
8. Fetch data with %SSC_MOVE.
9. Execute update statement with %SSC_EXECUTE (SSQL_STANDARD).
10. If there are more rows, fetch data (%SSC_MOVE) and then execute update (%SSC_EXECUTE) in loop.
11. Close cursor with soft close.
12. End transaction (%SSC_COMMIT).
13. If you want to rebind the SELECT statement, start a transaction, rebind, and loop through the first %SSC_MOVE, %SSC_EXECUTE, %SSC_MOVE, %SSC_SCLOSE, and %SSC_COMMIT to end the transaction.
14. Close cursor with soft close (%SSC_SCLOSE).
15. Release connection if no more operations (%SSC_RELEASE).



Insert

1. Initialize SQL Connection.
2. Connect to database.
3. Start transaction with %SSC_COMMIT.
4. Set up SQL statement.
5. Open cursor with SSQL_NONSEL.
6. Test for errors.
7. Set up data.
8. Execute statement with SSQL_STANDARD.
9. Loop until done.
10. End transaction by committing it or by rolling it back if there's an error.
11. Close cursor.
12. Release connection if no more operations.

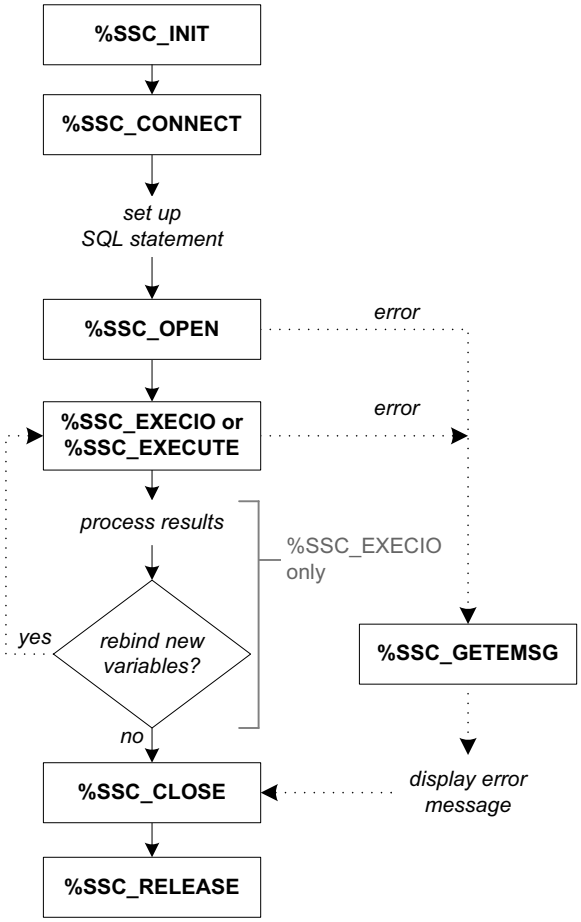
    For an example, see **exam_create_table.dbl**. See "Writing an SQL Connection Program" on page 2-2 for information on this program.



Stored Procedure

1. Initialize SQL Connection.
2. Connect to database.
3. Set up SQL statement.
4. Open cursor with SSQL_NONSEL.
5. Test for errors.
6. Set up bound data (%SSC_EXECIO only).
7. Execute stored procedure.
8. Test for errors.
9. Process any returned data (%SSC_EXECIO only).
10. Loop until done.
11. Close cursor.
12. Release connection if no more operations. For examples, see
    - **stp_mysql.dbl**
    - **stp_odbc.dbl**
    - **stp_ora.dbl**
    - **stp_sqlsrv1.dbl**
    - **stp_sqlsrv2.dbl**

    See "Writing an SQL Connection Program" on page 2-2 for information on these programs.



To fetch data from a SQL Server result set, use a combination of %SSC_OPEN with %SSC_MOVE (rather than %SSC_EXECIO or %SSC_EXECUTE) as illustrated in the diagram in "Multirow query" on page 2-11 and the **stp_sqlsrv1.dbl** and **stp_sqlsrv2.dbl** example programs.

Note that with stored procedures, transactions typically take place inside the stored procedure, not outside with %SSC_COMMIT.

Building Connect Strings

To connect to a database, you must pass a connect string to %SSC_CONNECT. A connect string contains the information needed to access a database. This differs for each database, but it generally includes information such as database name, user name, and password. The connect string also specifies whether SQL OpenNet will be used for the network layer.

There are two forms of connect string. The first, which starts with the *driver* argument, connects directly to a local database or database client. With this form, the database client must provide the network layer if necessary:

driver:*database_info*

The second, which starts with net:, uses SQL OpenNet for the network layer:

`net:`*database_info@opennet_info*

For example, the following uses SQL OpenNet to connect to an Oracle database on a Windows server (win_srv).

```
net:my_uid/my_pwd@win_srv!VTX0_10
```

For more examples, see "Driver and database_info notes and examples" on page 2-22.

Arguments

driver

The name of the database driver to be used. See "Driver and database_info" on page 2-18.

database_info

Information (user name, password, etc.) needed for database access. See "Driver and database_info" on page 2-18 for syntax.

opennet_info

Information needed to access an SQL OpenNet service on the machine that has the target database or database client. See "Network string (opennet_info) syntax" on page 2-19.

Discussion

When accessing a local database (i.e., a database or database client on the same machine as the SQL Connection program), use the first connect string form listed above, which connects directly to the database or database client. For a local connection there is generally no need for SQL OpenNet. (However, SQL OpenNet is needed for 32-bit SQL Connection applications on 64-bit Windows machines.) If you connect directly to a client for a remote database, the database network facilities (rather than SQL OpenNet) are used for the network layer, as illustrated in figure 2-1.

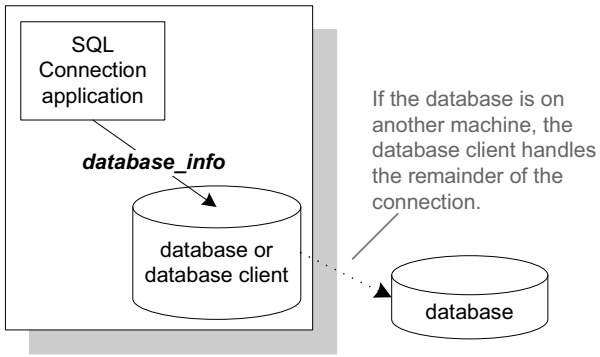


Figure 2-1. Connecting directly to a local database or database client.

When accessing a remote database, we recommend using the second connect string form (the form that starts with net:), which uses SQL OpenNet for the network layer. The second form connects to the SQL OpenNet server specified in *opennet_info* and uses *database_info* to access the database or database client on the server machine, as illustrated in figure 2-2.

There are two advantages to using SQL OpenNet for a connection over a network: it generally results in better performance, and the database driver for the target database does not need to be installed on each client (a requirement when connecting directly).

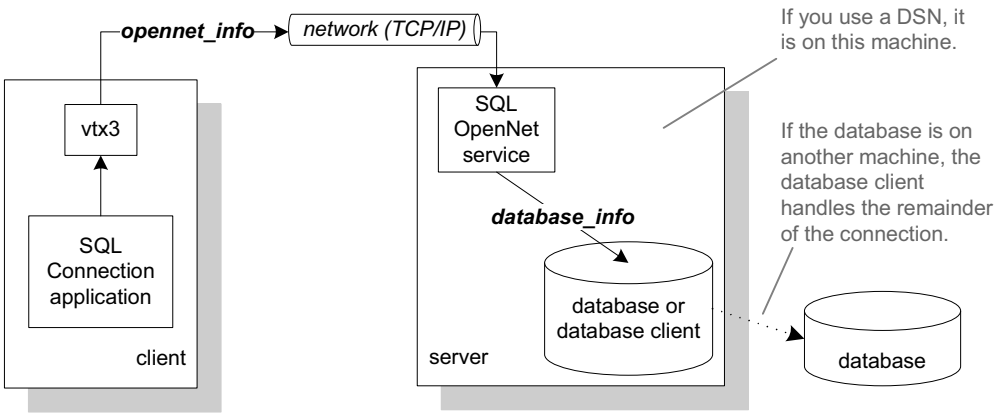


Figure 2-2. Using SQL OpenNet to connect to a database or database client.

Driver and *database_info*

The following table defines the *database_info* syntax for the different drivers.

***Driver* and *Database_info* Syntax**		
Database	***driver***	***database_info***
Oracle 9 on Windows	VTX0_9	*userid/password[/net_service_name]* See "Oracle notes and examples" on page 2-22.
Oracle 10 on Windows	VTX0_10	
Oracle 11 on Windows	VTX0_11	
Oracle on UNIX and OpenVMS	VTX0	
Oracle Rdb[a]	VTX1	*schema_name*
Sybase[a]	VTX2	*userid[/[password]/[database_name]/[server]/[appname]/[language][/charset]]*
SQL OpenNet	VTX3	n/a. This is the SQL OpenNet client. It is used when the connect string starts with net:.
Synergy Database	VTX4	*userid/[password]*/sdms:*connect_file*
Informix[a]	VTX5	*database_name*
ODBC-compliant[a]	VTX11	*userid/[password]/dsn* See "VTX11 (ODBC) notes and examples" on page 2-23.
SQL Server	VTX12_ODBC (ODBC)	*userid/[password]/dsn* or *userid/[password]/[database_name]/[server_name[\\instance_name]]/[app_name]/[language][/other_options]* See "SQL Server notes and examples" on page 2-23.
	VTX12_SQLNATIVE (SQL Native Client)	
MySQL	VTX14	*userid/[password]/[database]/[server]* See "MySQL connect string examples" on page 2-25.

a. Support for these databases may require assistance from Synergex Professional Services and additional support fees. Contact your Synergy/DE account manager for details. See "Synergex Professional Services Group" on page ix.

Note the following for *driver*:

- On Windows, a database driver consists of a DLL and an executable, which are located in the synergyde\connect directory. (The *driver* argument represents both.) Do not include a path for *driver*—just the driver name itself.
- On UNIX, *driver* is a shared object (an **.so** file) that is installed in the synergyde/connect directory. *Driver* can include a path on UNIX, but there's no need to specify one unless you move the shared object to a directory other than synergyde/connect.
- On OpenVMS, *driver* is a database driver (an **.exe** shared image file) that is located in the SYS$COMMON:[SYSLIB] directory. Do not include a path for *driver*—just the driver name itself.

Network string (*opennet_info*) syntax

The *opennet_info* argument (see "Arguments" on page 2-16) provides the information needed to connect to an SQL OpenNet service on a machine that has either the database or a client for the database. Here is the syntax:

@[port:]host[([domain\]uid/pwd)]!driver[,ENV_VAR=env_spec,…]

Arguments

port

(optional) The port number for communicating with SQL OpenNet server. Specify this only if you want to override the default, which is generally 1958. See "Port settings" on page 2-21.

host

The IP address or name of the machine that has the database or database client. Note that server names can be up to 64 characters long.

([domain\]uid/pwd)

(optional) Login information (user ID and password) for an account on the host machine or, if *domain* is also specified, an account on a domain (Windows only). Note that this is different than the user ID and password for the database, which are passed as *database_info* in the connect string.

driver

The name of the database driver to be used. See "Driver and database_info" on page 2-18.

ENV_VAR=env_spec

(optional) An environment variable definition stored in or used by the target database (generally for specifying or locating data files).

Discussion

The network string (*opennet_info*) is the part of a connect string that starts with an "at" sign (@). It is included only when SQL OpenNet is used (i.e., when the connect string starts with net:). In the following, for example, the bold text is the network string (win_srv is *host*, and VTX12_SQLNATIVE is *driver*):

```
net:my_uid/my_pwd/my_dsn@win_srv!VTX12_SQLNATIVE
```

For more examples, see "Driver and database_info notes and examples" on page 2-22.

Note the following for *domain*, *uid*, and *pwd*:

- Use these only if the **-a** option is specified for **vtxnetd** or **vtxnet2**. (See "The vtxnetd and vtxnet2 Programs" in the "Configuring Connectivity Series" chapter of the *Installation Configuration Guide* for information.)
- The parentheses and slashes are required for these arguments—e.g., (my_domain\my_uid/my_pwd). Use the backslash only if you specify a Windows domain.

Note that when using **vtxnet2** to access a database on a Windows server, the "Log on as a batch job" option must be set for the user account specified by *uid*. Domain user accounts must have a group policy that includes the "Log on as a batch job" option.

Overriding delimiters in a network string

SQL OpenNet uses special characters as string delimiters: the at sign (@), colon (:), and exclamation point (!). In the network part of a connect string, each of these delimiters conveys a specific instruction to the SQL OpenNet processor and generally is *not* passed by the processor unless an identical character follows the first. If you want to pass an at sign, colon, or exclamation point as part of an environment variable definition, or at any other place in the network connect string, you must use a duplicate at sign, colon, or exclamation point to ensure that the parser will interpret the statement correctly.

For example, @unix_srv in the following connect string is the host name of the computer containing the database.

```
net:my_uid/my_pwd@@unix_srv!/usr/synergyde/connect/VTX0
```

In the following, DBDATA is set to datdir!.

```
net:my_uid/my_pwd@unix_srv!/usr/synergyde/connect/VTX0,DBDATA=datdir!!
```

The following connect string uses two "at" signs to force the first to become the delimiter for a network string for a second server. (See the Synergex KnowledgeBase article 100002075 for more information on this configuration.)

```
net:my_uid/my_pwd@88.0.0.12!/usr/bin/VTX3@@unix_srv2!/bin/VTX0
```

Port settings

To make a connection with SQL OpenNet, the port setting on the client must match the port number for the SQL OpenNet server. For example, if **vtxnetd** is started on port 1990, the client must use port 1990 to connect to it.

On Windows and UNIX servers, you can specify the port number in

- the vtxnet setting in the TCP/IP **services** file, which is in %windir%\system32\drivers\etc on Windows and /etc on UNIX. This is the default port setting, but it is used only when the port is not specified in the **vtxnetd** or **vtxnet2** start-up command.
- the **vtxnetd** or **vtxnet2** command line in **opennet.srv** (on Windows) or the **vtxnetd** command line in the **startnet** script (on UNIX). This overrides the **services** file setting.

On an OpenVMS server, the default (1958) is hard-coded, but you can override that by setting a port number in the **vtxnetd** command line in **NET.COM**.

For more information on server-side port settings, see the "Configuring Connectivity Series" chapter of the *Installation Configuration Guide*.

On clients, you can specify the port number in

- the vtxnet setting in the TCP/IP **services** file, on Windows (%windir%\system32\drivers\etc) and UNIX (/etc). This is the default port setting, but it is used only when there is no other port setting on the client. For more information, see the "Configuring Connectivity Series" chapter of the *Installation Configuration Guide*.
- a port setting in **net.ini**. If set here, this overrides the **services** setting only. (Connect string settings override a port setting in **net.ini**.) For more information, see "Setting connect string defaults and encryption in net.ini" on page 1-7.
- a port setting in the connect string. If set here, this overrides all other port settings for the client.

A quick way to ensure your port settings match is to use either the **synxfpng** utility (with the **-x** option) or the **vtxping** utility without specifying a port in the command line. If the connection is successful, the port settings match. For details, see "The vtxping Utility" in the "Configuring Connectivity Series" chapter of the *Installation Configuration Guide* and "The synxfpng Utility" in the "Configuring xfServer" chapter of the *Installation Configuration Guide*.

Driver and *database_info* notes and examples

The following sections discuss arguments for the *database_info* portion of a connect string (see "Driver and database_info" on page 2-18) and provide examples for several of the supported databases/drivers.

Oracle notes and examples

The *net_service_name* argument in *database_info* specifies a database instance and is required if the database has multiple instances. Note that the spelling of the database instance name must match the spelling used in the SQL*Net configuration file **TNSNAMES.ORA**. As an alternative to the *net_service_name* argument, you can use the *ENV_VAR* argument documented in "Network string (opennet_info) syntax" on page 2-19 to set the Oracle system ID (SID) for the instance. See Oracle documentation for more information.

We recommend using SQL OpenNet for network connections, but you can connect directly to an Oracle database via the Oracle client. If you do, note the following:

- Oracle will use SQL*Net to make the connection, so the connect string must use SQL*Net syntax. (See Oracle documentation.) For example, the following connects to an Oracle 11 server named **oracle_srv**:

  ```
  VTX0_11:my_uid/my_pwd/oracle_srv
  ```

- You must first run the Oracle script **oraenv** to connect directly to an Oracle database on UNIX.

We recommend against recycling database connections, See "Database connections" on page 2-5.

Connect string examples for Oracle

The following connects directly to a local Oracle 10 database or database client on a Windows machine. *Database_info* is my_uid/my_pwd.

```
VTX0_10:my_uid/my_pwd
```

The next connect string uses SQL OpenNet to connect to a Windows server (win_srv). No port number is specified, so the default is used.

```
net:my_uid/my_pwd@win_srv!VTX0_10
```

The next uses SQL OpenNet to connect to a UNIX server. Note that the connect string includes a path for the database driver (VTX0). This is necessary if the database driver is not in the synergyde/connect directory.

```
net:my_uid/my_pwd@unix_srv!/usr/my_dir/connect/VTX0
```

The next example also uses SQL OpenNet to connect to a UNIX server. Host_uid/host_pwd is used to log on to the server (unix_srv), and my_uid/my_pwd is used to log on to the database on the server. Again, because the database driver isn't in the synergyde/connect directory, the connect string includes a path for the database driver.

```
net:my_uid/my_pwd@unix_srv(host_uid/host_pwd)!/usr/mydir/connect/VTX0
```

The following uses SQL OpenNet to connect to an OpenVMS server.

```
net:my_uid/my_pwd/my_instance.com@vms_srv!VTX0
```

VTX11 (ODBC) notes and examples

To access a database using the ODBC database driver (VTX11), a user or system DSN must be defined for the database. For SQL OpenNet connections, the DSN must be on the server. For direct connections, the DSN must be on the clients. See Microsoft documentation for information on DSNs.

Connect string examples for VTX11

The following example connects directly to an ODBC-compliant database (in other words, it doesn't use SQL OpenNet). *Database_info* is my_uid/my_pwd/my_dsn. Information on the database is in the DSN (my_dsn).

```
VTX11:my_uid/my_pwd/my_dsn
```

The next example uses SQL OpenNet to connect to a Windows server (win_srv). *Database_info* is my_uid/my_pwd/my_dsn.

```
net:my_uid/my_pwd/my_dsn@win_srv!VTX11
```

SQL Server notes and examples

The Connectivity Series distribution includes two database drivers for SQL Server:

- VTX12_SQLNATIVE, which uses the Microsoft SQL Native Client ODBC driver
- VTX12_ODBC, which uses the ODBC API

For best performance, we recommend using VTX12_SQLNATIVE. And note that you must use it to use shared memory access (which we recommend whenever you use SQL OpenNet; see "Using the SQL Server shared memory protocol" on page 2-26).

For SQL Server, there are two forms of syntax for the *database_info* section of the connect string (see "Driver and database_info" on page 2-18):

- If the *database_info* section has two forward slashes, the first form (*userid/[password]/dsn*) is used to interpret the connect information.
- If there are more than two forward slashes, the second form is used. The second form enables you to create DSN-less connections.

The *other_options* part of the second form of *database_info* represents a string that can contain SQLDriverConnect() options that are specific to SQL Server. (See Microsoft documentation for information.) This string can contain multiple options separated by semicolons (;). The following connect string, for example, uses the SQL Server shared memory protocol (Network=dbmslpcn) and passes a DSN specification (DSN=my_dsn):

```
VTX12_SQLNATIVE:my_uid/my_pwd/////Network=dbmslpcn;DSN=my_dsn
```

For DSN-less connections, you can also pass Driver={*odbc_driver*} as *other_options*, where *odbc_driver* is a driver name returned by SQLDrivers(). (Without this setting, SQL Connection uses an ODBC driver of its own choosing.) The following uses SQL Server Native Client 11.0:

```
VTX12_SQLNATIVE:uid/pwd/pubs/srv///Driver={SQL Server Native Client 11.0}
```

We recommend using DSNs because they include client configuration information that would otherwise need to be set separately. We also recommend making DSN access local when you set up the SQL Server database instance.

- For a SQL OpenNet connections, DSNs must be system DSNs on the server (see figure 2-2 on page 2-17). For direct connections, DSNs must be user or system DSNs on the clients.
- For VTX12_ODBC, DSNs must be configured for ODBC. For VTX12_SQLNATIVE, DSNs must be configured for one of the SQL Native Client versions.

Unless you're using Windows authentication, make sure your SQL Server database is set up to use SQL Server authentication. For example, for SQL Server 2012 you can use SQL Server Management Studio to set the "SQL Server and Windows Authentication mode" option.

For more information on connections to SQL Server, see "Using your program with different drivers and databases" on page 2-7.

Connect string examples for SQL Server

The first example below illustrates the first form of *database_info* syntax: (*userid/[password]/dsn*). It creates a direct connection (i.e., SQL OpenNet is not used), and the information in the DSN determines whether the database is local or remote.

```
VTX12_ODBC:my_uid/my_pwd/my_dsn
```

The next example is also for a direct connection, but it uses the second form of *database_info* syntax. Note that it continues on a second line, and note that the DSN (my_dsn) is passed in the *other_options* part of *database_info*. The connection will use the SQL Server shared memory protocol (which improves performance) because the connect string includes Network=dbmslpcn. It will also use multiple active result sets (MARS) because the connect string sets MARS_Connection=yes.

```
VTX12_SQLNATIVE:my_uid/my_pwd/my_db////Network=dbmslpcn;DSN=my_dsn;
MARS_Connection=yes
```

The next example is also for a direct connection, uses the second form of *database_info*, and continues on a second line. But with this example, no DSN, user ID, or password is specified. Instead, the connect string instructs SQL Server to use Windows authentication (Trusted_connection=yes) for the user ID and password, and the SQL Server driver is specified by passing Driver={SQL Server Native Client 10.0}. Both are passed as *other_options*. My_db is the database name, and my_instance is the instance name. The period before my_instance indicates that the instance is on the local machine.

```
VTX12_SQLNATIVE://my_db/.\\my_instance///
Driver={SQL Server Native Client 10.0};Trusted_connection=yes
```

The following also specifies an instance name (TESTDB):

```
vtx12_sqlnative:sql_user/sql_user//MY_SERVER\\TESTDB
```

The following example uses SQL OpenNet to connect to a Windows server. *Database_info* is my_uid/my_pwd/my_dsn, *port* is 1960, and *host* is win_srv.

```
net:my_uid/my_pwd/my_dsn@1960:win_srv!VTX12_ODBC
```

The next example, which is continued on a second line, uses SQL OpenNet to connect to a port number 1960 on a Windows server (win_srv).

```
net:my_uid/my_pwd/my_db////Network=dbmslpcn;DSN=my_dsn;
MARS_Connection=yes@1960:win_srv!VTX12_SQLNATIVE
```

MySQL connect string examples

Connections to MySQL use the MySQL database driver, VTX14.

The following connect string is for a direct connection (i.e., it doesn't use SQL OpenNet). *Database_info* is my_uid/my_pwd/my_db.

```
VTX14:my_uid/my_pwd/my_db
```

This next example uses SQL OpenNet to connect to a Windows server (win_srv). *Database_info* is my_uid/my_pwd/my_db.

```
net:my_uid/my_pwd/my_db@win_srv!VTX14
```

The next example uses SQL OpenNet to connect to a Linux server (linux_srv). Because the database driver isn't in the synergyde/connect directory, the connect string includes a path for the driver.

```
net:my_uid/my_pwd/my_db@linux_srv!/usr/my_dir/connect/VTX14
```

Using the SQL Server shared memory protocol

For improved performance for connections to SQL Server, use the SQL Server shared memory protocol. This reduces the number of TCP/IP sockets used for a connection, making it less likely that all of the sockets for the server machine will be used at one time, which can greatly impede performance. Note the following:

- For remote databases, the SQL Server shared memory protocol is available only if you use VTX12_SQLNATIVE and if you use SQL OpenNet for all network connections. So, if you use a DSN, for example, the DSN must be on the machine that has the SQL Server database.
- For connections that use DSNs, select "(local)" as the server name in the DSN configuration screen, and prefix this with "LPC:" (without quotes)—i.e., LPC:(local). See figure 2-3 below.

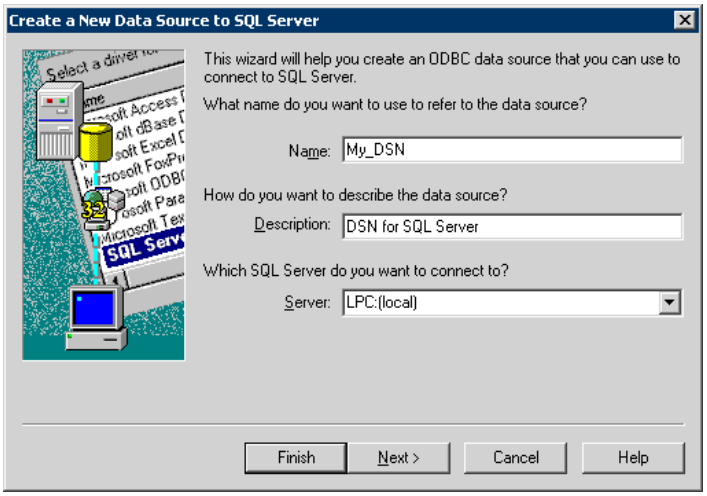


Figure 2-3. Specifying LPC:(local) as the server for a DSN when using VTX12_ODBC.

- For DSN-less connections, use the second form of syntax for *database_info* and either omit the server name from the *database_info* string or add "Network=dbmslpcn" to the *other_options* string. (This setting specifies the shared memory network library for SQL Server.) For example, the following establishes a connection to a local database:

  ```
  VTX12_SQLNATIVE:my_uid/my_pwd/my_db////Network=dbmslpcn
  ```

  The next example uses SQL OpenNet to connect to a remote database:

  ```
  net:my_uid/my_pwd/my_db////Network=dbmslpcn@win_srv!VTX12_SQLNATIVE
  ```

Cursors

SQL Connection uses two types of cursor: *database cursors* and a special SQL Connection mechanism called *logical cursors*. A database cursor is a processed SQL statement (one that the database has parsed, optimized, and so forth—i.e., a cached execution plan) and/or the database mechanism for traversing and maintaining a position on a row in the result set. Logical cursors, on the other hand, are SQL Connection mechanisms for accessing cursors, including soft-closed cursors. You'll allocate logical cursors when you initialize a database channel (%SSC_INIT), but otherwise, consider all discussions of cursors in this manual to be discussions of database cursors. (Under the hood, however, SQL Connection uses logical cursors to streamline data access.)

With SQL Connection, you interact with cursors by

- opening a new cursor or reusing a cursor when you pass an SQL statement to %SSC_OPEN.
- optionally using %SSC_CMD to set options for subsequently opened cursors.
- passing the cursor ID to other SQL Connection routines that require it to access data or update the database (%SSC_BIND, %SSC_MOVE, and so forth).
- hard-closing or soft-closing the cursor.

Closing cursors

Reusing cursors can improve performance, but cursors take up resources that you'll generally want to release as soon as possible. So SQL Connection includes two ways to close cursors: soft closing (%SSC_SCLOSE) and hard closing (%SSC_CLOSE).

- Soft closing frees some database resources on SQL Server, but on other databases it increases the chance that the database will retain the cached results, if results are still pending for the operation. Soft closing may enable the database to cache the execution plan and just rebind variables. Results depend on your database.

  In general, consider soft closing the cursor (by calling %SSC_SCLOSE) if your program will redo the same operation soon or frequently and if your program just retrieves one row or keeps retrieving until the database has no more data for the statement. See "Reusing cursors" on page 2-28 for more information.

- Hard closing frees all memory for reuse and frees database resources associated with the cursor, including removing the cached execution plan and all locks, closing the database cursor, and so forth. In general, consider hard closing the cursor (by calling %SSC_CLOSE) if you know that your program will not reuse a statement soon or frequently, or if no more data will be retrieved for the statement. Additionally, note that %SSC_COMMIT, %SSC_RELEASE, and %SSC_ROLLBACK will also hard close cursors.

Because databases hold a cache of previously used statements, a database cache may reach its limit (resulting in a severe decrease in performance) if you do not hard close.

.NET

Note that with Synergy .NET in a multi-threaded environment, cursors are not automatically closed when threads terminate.

Reusing cursors

For optimal performance, reuse a cursor if the statement will be reused soon or frequently. Cursor reuse can significantly improve database and network performance. It saves time opening the cursor, and it uses less memory. SQL Connection enables you to reuse cursors if

- the database allows cursor reuse.
- the cursor is still open or has been soft-closed.
- the SQL statement in %SSC_OPEN is identical in every way to the SQL statement previously used with the cursor.

In other words, reuse a cursor if you are processing the same SQL statement several times with the same or different bind data.

In brief, these are the steps that an SQL Connection application and the database take to process an SQL statement if you don't reuse a cursor:

1. Open cursor.
2. Process the SQL statement:
   - Check cache. Is there an identical statement in the cache? If so, use cached entry and go to step 3.
   - Parse. Is the statement correct? Does the syntax make sense?
   - Bind. Do specified data objects (tables and columns) exist?
   - Check authorization. Is the user allowed to access the data?
   - Plan access. How is this data to be accessed?
   - Optimize. How can data be retrieved more efficiently?
3. Bind parameters.
4. Execute the statement (e.g., fetch data).
5. Close the cursor.

When a cursor is reused, however, the application skips the initial step of processing the SQL statement. This alone saves a great deal of overhead since initial processing is typically a very expensive process, using as much as 10 times the resources as other steps. The reused cursor rebinds only the variables containing new data and eliminates the need for re-parsing the entire statement. You can then fetch new data and rebind the host variables as many times as necessary.

SQL Connection reuses cursors in conjunction with the %SSC_OPEN, %SSC_SCLOSE (as mentioned in "Closing cursors" above), %SSC_REBIND, %SSC_EXECUTE, and %SSC_EXECIO functions. For an example, see the example section for %SSC_CLOSE on page 3-6.

Note the following:

- When using SQL statements with conditions, use bind variables (not literals) for the conditions to ensure that the SQL statement matches the SQL statement originally used with the cursor.
- Avoid using string arithmetic to build SQL statements. Instead, use literals.
- Be sure to explicitly end transactions. If you do not explicitly commit or roll back, it's up to the database to determine how to end the transaction, which may lead to unexpected results, including running out of resources.
- You can use Vortex API logging to find out how well you've optimized cursor usage. See "Using Vortex API logging to verify optimization" on page 5-7 for information.
- If you are fetching a row and you plan to perform a positioned update, you can use %SSC_SQLLINK to link the update statement to the open SELECT cursor rather than opening another cursor for the update statement. See %SSC_SQLLINK on page 3-56 for more information.

Cursor types

The *options* argument for %SSC_OPEN and some of the %SSC_CMD options enable you to create forward-only, dynamic, keyset-driven, and static cursors, though not all databases support all of these types. Additionally, in some cases a cursor can be a scrolling cursor, which enables you to determine which row will be retrieved with the next fetch (by setting an SSQL_CMD_SCROLL option in a call to %SSC_CMD). Non-scrolling cursors, on the other hand, always retrieve the next row. Supported cursor types are briefly discussed below, but see your database documentation for more information, and see "Specifying a cursor type" on page 2-31 for information on how to select the cursor type for a statement. Note the following:

- To determine which type of cursor works best for your statements, perform timing tests and examine the database performance logs.
- If you change the SQL Connection cursor type when reusing a cursor (in other words, when passing a cursor number in the *dbcursor* argument for %SSC_OPEN), the cache for cursor reuse for identical statements is flushed.

Forward-only. Forward-only cursors are generally the most efficient cursors for read operations because for some databases, an entire result set may be cached on the client or in a network buffer. (Changes made to the database after the result set is established are not reflected in the result set.) Additionally, forward-only cursors

- are the only type of cursor most databases support. This is the default setting for all but VTX12_ODBC and VTX12_SQLNATIVE (which have a default setting of dynamic).
- may support updates, except when using VTX12_ODBC or VTX12_SQLNATIVE. Note that forward-only cursors don't support positioned updates when using VTX11.
- cannot be scrolling cursors.
- are client-side cursors when accessing a SQL Server database.
- enable you to have multiple concurrently active statements. Note, however, that this may not be true if you are accessing SQL Server with VTX12_ODBC or VTX12_SQLNATIVE. For these drivers, forward-only cursors generally don't support multiple concurrently active statements because SQL Server returns the "default result set" (also known as a firehose cursor) when a cursor is set to forward-only (and SSQL_STANDARD, an %SSC_OPEN option). This default result set must be processed before another statement can be issued, which limits the client to one active SQL statement at a time. (If you attempt to issue an SQL statement while a previous statement is still active, you may get a "Connection is busy with results for another hstmt" error.) However, when using multiple active result sets (MARS) with SQL Server, this may not be a limitation.

Dynamic. Dynamic (also known as sensitive) is a SQL Server cursor type that reflects all changes to the database made by other users. A dynamic cursor may be better than a forward-only cursor for a large result set if only part of the result set will be read, or if the result set is too large for the network buffer used for forward-only cursors. Additionally, dynamic cursors

- can be used only with VTX11 (if the database supports them), VTX12_ODBC, and VTX12_SQLNATIVE. It is the default setting for VTX12_ODBC and VTX12_SQLNATIVE (for application compatibility with other databases). However, we recommend using the forward-only setting when possible. See "Specifying a cursor type" on page 2-31.
- are the only cursors you can use to update data or delete rows when using VTX12_ODBC or VTX12_SQLNATIVE.
- can be scrolling cursors.
- are server-side cursors.
- enable you to have multiple concurrently active statements.

Keyset-driven. This is a SQL Server cursor type that is updated only with changes made to rows that existed when the statement was executed (not rows inserted after the result set was established). Keyset-driven cursors are usually the least efficient cursors. Additionally, keyset-driven cursors

- can be used only with VTX11 (if the database supports them), VTX12_ODBC, or VTX12_SQLNATIVE.
- don't support updates.
- create a temporary table in the temporary database.
- can be scrolling cursors.
- are server-side cursors.
- enable you to have multiple concurrently active statements.

Static. Static (also known as insensitive) is a SQL Server cursor type that does not reflect any changes made to the database after the result set was established. Static cursors are the most efficient scrolling cursors. Additionally, static cursors

- can be used only with VTX11 (if the database supports them), VTX12_ODBC, or VTX12_SQLNATIVE.
- don't support updates.
- create a temporary table in the temporary database.
- can be scrolling cursors.
- are server-side cursors.
- enable you to have multiple concurrently active statements.

Specifying a cursor type

To specify the type of cursor you want, use cursor options for %SSC_OPEN, %SSC_CMD, or both.

To create a *forward-only* cursor (a static cursor that cannot be made to scroll), call %SSC_CMD with the SSQL_CURSOR_FORWARD_ONLY option (or the SSQL_CURSOR_DEFAULT option) before calling %SSC_OPEN. Then make sure the %SSC_OPEN call does not include any of the SSQL_SCROLL options. Note the following:

- For all database drivers except VTX12_ODBC and VTX12_SQLNATIVE, the default cursor type is forward-only, so you can just omit any %SSC_CMD or %SSC_OPEN option that would set a type. (For compatibility with other databases, the default setting for VTX12_ODBC and VTX12_SQLNATIVE is dynamic.)
- For VTX12_ODBC and VTX12_SQLNATIVE, using %SSC_CMD to set SSQL_CURSOR_DEFAULT (or SSQL_CURSOR_FORWARD_ONLY) without setting any other cursors options (for example, scrolling options) causes SQL Server to return the default

result set, which is also known as the firehose cursor. Generally, the default result set is faster than other cursor types for SQL Server, but there are some limitations. See the note under "Forward-only" on page 2-30.

- For SQL Server, you can also create a forward-only server-side cursor by specifying the %SSC_CMD command SSQL_RO_CURSOR when specifying dynamic, keyset-driven, or static cursors. Note, however, that in this case, SQL Server will not return the default result set (the firehose cursor). You must use %SSC_CMD to set SSQL_CURSOR_DEFAULT (or SSQL_CURSOR_FORWARD_ONLY) as mentioned above.

To create a *dynamic*, *keyset-driven*, or *static* cursor, do one of the following:

- Use the SSQL_CURSOR_TYPE options in a call to %SSC_CMD and then, if you want a scrolling cursor, pass SSQL_SCROLL in the %SSC_OPEN call. (Note that to make a cursor scrollable, you must use one of the SSQL_SCROLL options for %SSC_OPEN.)
- Without setting any SSQL_CURSOR_TYPE options (for %SSC_CMD), call %SSC_OPEN with one of the SSQL_SCROLL options.

For VTX12_ODBC and VTX12_SQLNATIVE,

- if you use static or dynamic cursors and your application does not update the database, we recommend setting the SSQL_RO_CURSOR option (an %SSC_CMD option). This option instructs SQL Connection to use fast-forward cursors (SQL_CO_FFO) when possible.
- watch for implicit conversions, which adversely affect performance and indicate that you are using the wrong cursor type for the statement. To detect implicit cursor conversions, use SQL Connection logging (see "SQL Connection logging" on page 5-5). For example, if you are using dynamic or keyset-driven cursors, SQL Server will generate "Success with info" messages when it automatically changes cursor type.
- SQL Server uses pessimistic locking when using dynamic cursors and the (UPDLOCK ROWLOCK) hint. See "Row locking" on page 2-42.
- The default concurrency setting for a cursor is SQL_CONCUR_READ_ONLY. However, when you use SSQL_POSITION or SSQL_FORUPDATE, VTX12_ODBC uses SQL_CONCUR_LOCK.

See Microsoft Developer Network (MSDN) documentation for more information on these topics.

Data Mapping

As you add SQL statements to your program, you can use variables to store data sent to and received from the database. This is called data mapping. *Defined* variables store data received from a database. *Bind* variables store data that's sent to a database.

Since SQL Connection always attempts to convert data for given types, make sure you use the proper data types for both defining and binding. In addition, remember that data conversion is a database-specific operation. See "Data Conversion" on page 2-36 for the recommended use of data types.

Defining variables

If your SQL Connection program includes an SQL statement that returns data, you must use defined variables to store the returned data. To do this, declare a variable for each column that will be returned from the query. Then, in a call to %SSC_DEFINE, specify the declared variables in the order that their corresponding columns are specified in the SQL statement. %SSC_DEFINE maps the variables to the returned columns of data. Finally, call %SSC_MOVE to fetch the data into the variables. Note that you may want to use %SSC_STRDEF rather than %SSC_DEFINE if you are passing many variables. Depending on the number of variables you pass, %SSC_STRDEF may improve performance. For more information, see the following:

- %SSC_DEFINE on page 3-21
- %SSC_MOVE on page 3-41
- %SSC_STRDEF on page 3-58

The following example illustrates how you can handle an SQL statement that returns just one row of data:

```
sqlp="SELECT deptnum FROM org1 WHERE division=:1"
if (%ssc_open(dbchn, cur1, sqlp, SSQL_SELECT,, 1, a_bind_var))
  goto err_exit
if (%ssc_define(dbchn, cur1, 1, a_deptnum))    ;Map variable to column.
  goto err_exit
sts = %ssc_move(dbchn, cur1)            ;Fetch data into defined variable.
```

If the SQL statement returns more than one row of data, put the %SSC_MOVE call in a loop. For example:

```
sqlp="SELECT deptnum, deptname, hrdate, salary FROM org1
;     WHERE division=:1"
if (%ssc_open(dbchn, cur1, sqlp, SSQL_SELECT,, 1, a_bind_var))
  goto err_exit
if (%ssc_define(dbchn, cur1, 4,a_deptnum, a_deptname, a_hrdate,
;               a_salary))              ;Map variables to columns.
  goto err_exit
```

```
do forever
  begin
    sts = %ssc_move(dbchn, cur1, 1, rowcnt)     ;Fetch a row of data into
.                                               ; defined variables.
.
.
```

Binding data

For data that's sent to the database, you can use bind variables to bind the data, rather than hard-coding the data in the SQL statement. You can bind

- data that updates the database—for example, data in the VALUES clause of an INSERT statement or the SET clause of an UPDATE statement.
- data that's part of the selection criteria—for example, data in a WHERE or LIKE clause.

Binding enables the database to prepare an SQL statement once and then reuse the prepared statement as many times as your program sends new data for the statement. (For information on resubmitting a statement after updating bind variables, see %SSC_REBIND on page 3-49.)

When you bind data, you map the data with a one-to-one mapping method (one Synergy DBL variable holds one column of data in the database row) on a column-by-column basis.

Binding takes place when the query is executed, which for SELECT statements happens with %SSC_OPEN and for non-SELECT statements happens with %SSC_EXECUTE or %SSC_EXECIO. For example, if you bind the host variable ordnum to a SELECT statement and ordnum equals 2 when %SSC_OPEN is called, the SELECT statement will use 2 for the query even if the ordnum value is changed to 3 before %SSC_MOVE is called. However, for a non-SELECT statement, the value 3 would be used because the variable ordnum is evaluated by %SSC_EXECUTE, which would follow %SSC_MOVE.

Specifying bind variables

To use a bind variable, put a placeholder (described below) in the SQL statement and pass a bind variable in an %SSC_OPEN, %SSC_BIND, or %SSC_SQLLINK call.

- With %SSC_OPEN, you can bind variables for SELECT and non-SELECT statements.
- With %SSC_BIND and %SSC_SQLLINK, you can bind variables only for non-SELECT statements.
- If you have linked a statement to a SELECT cursor (using %SSC_SQLLINK), a subsequent call to %SSC_STRDEF binds variables for the original SELECT statement, and a subsequent call to %SSC_BIND binds variables for the linked statement.

For example, the following INSERT statement has six placeholders in the VALUES clause, and the %SSC_OPEN call has six bind variables that correspond to the placeholders: ":1" is a placeholder for deptnum, and ":2" is a placeholder for deptname, and so forth. If necessary, the statement could also have a WHERE clause with additional bind variables.

```
    sqlp = "INSERT INTO ORG1 (DEPTNUM, DEPTNAME, MANAGER, DIVISION, "
&    "HRDATE, SALARY) VALUES (:1,:2,:3,:4,to_date(:5,'MM/DD/YYYY'),:6)"

    if (%ssc_open(dbchn, cur1, sqlp, SSQL_NONSEL, SSQL_STANDARD, MX_VARS,
&                 deptnum, deptname, manager, division, hrdate, salary))
      goto err_exit
```

For Oracle databases, the placeholder numbers determine the order in which the variables are used. For other databases, bind variables are used in the order they are specified in the %SSC_OPEN, %SSC_BIND, or %SSC_SQLLINK call; the placeholder number does not affect the order for these databases, but it is required. Note the following:

- Do not duplicate a number.
- Use consecutive numbers starting with 1.

Note that you can also use %SSC_STRDEF to bind (and define) variables for SELECT and non-SELECT statements. When possible use %SSC_OPEN, %SSC_BIND, or %SSC_SQLLINK, but for SELECT statements, you cannot bind more than 248 variables unless you use %SSC_STRDEF. And for non-SELECT statements, you cannot bind more than 256 variables unless you use %SSC_STRDEF or use multiple %SSC_BIND calls with 256 or less bind variables in each call.

Also note that if you use a bind variable to hold an operand for a LIKE clause, be sure to use a database trim function to trim trailing spaces. (For example, use RTRIM with SQL Server or TRIM with Oracle.) If you don't, the trailing spaces will be evaluated as part of the LIKE clause. For example, if the bind variable is an **a10**, for example, and the LIKE clause operand is "a%", the LIKE will specify a match for a% plus eight trailing spaces. In this case, for example, "anderson" would be overlooked.

Data Conversion

SQL Connection automatically converts Synergy data types to database-specific data types when data flow is from a Synergy application to the database. And database-specific data types are converted into Synergy data types when data flow is from the database to a Synergy application. Note the following:

- For information on binding and defining large binary or character columns, see the %SSC_LARGECOL Discussion on page 3-39.
- If you store implied-decimal values as whole numbers (programmatically handling the decimal point), you must use the ^D() function to convert variables that store the whole numbers into the correct implied-decimal format for sending data to and receiving data from the database. If you don't, only the whole number part of a decimal value will be stored. For example, if you use a **d10** variable (instead of a **d10.2**) for a currency field, the value 100.02 will be truncated to the whole number 100. For information on ^D(), see ^D in the "System-Supplied Subroutines and Functions" chapter of the *Synergy DBL Language Reference Manual*.
- Date-to-numeric conversions result in Julian date values that are compatible with %NDATE.

Data conversion when binding

When binding host variables to database columns, SQL Connection makes the following data conversions:

Binding Synergy DBL host variables...	...to database columns
Alpha	Binary, char, date, datetime, varchar
Decimal	Numeric
Implied decimal	Float, numeric
Integer	Integer, numeric
System.String	Binary, char, date, datetime, varchar

Note the following:

- For alpha host variables, fields that start with a binary 0 become null. Depending on the database, data for alpha host variables may be converted in other cases as well. For example a blank field or a field filled with spaces may be converted to null or to a single space. It may also remain as is. And a field with trailing blanks may be trimmed for varchar. For information on an option that controls the way Oracle treats data for alpha host variables, see SSQL_TRIMCHAR on page 3-15.
- For decimal and implied decimal host variables, blanks become zeros, ASCII zeros remain zeros, and host variables that start with binary zero (see note below) become null values.

- For integer host variables, binary zeros remain binary zeros. There is no way to store a null value using a bound integer field.
- For binary columns with SQL Server or Oracle, %SSC_EXECIO treats the data as a char field and trims trailing spaces, unless you can use the SSQL_EXBINARY option. (If you use the SSQL_EXBINARY option, %SSC_EXECIO uses the given data and length.) %SSC_MOVE always preserves binary column data.
- To bind a null value to a char, date, datetime, numeric, or float database column, set the first character position of your alpha or decimal field to binary zero with %CHAR(0). If you are using a decimal field, you will need to use the ^A() function to cast it as an alpha. (You cannot store a null value using a bound integer field.) See "Using %SSC_INDICATOR when updating a column with null" on page 2-39 for a method for doing this, and remember to reset the column to its original value before using it with anything other than an %SSC_ function.

  For more information about ^A and %CHAR, see ^A and %CHAR in the "System-Supplied Subroutines and Functions" chapter of the *Synergy DBL Language Reference Manual*.
- For non-array operations, SQL Connection can convert data in System.String bind variables to char, date, datetime, or varchar and, when using %SSC_LARGECOL, to binary. (System.String bind variables are not supported for array-based operations.)

Data conversion when defining

When loading database columns to defined host variables, SQL Connection makes the following conversions:

Loading database columns...	...to defined Synergy DBL host variables
Binary	Alpha, System.String
Char	Alpha, System.String
Currency	Implied decimal
Date	Alpha, decimal, integer, System.String
Datetime (including datetime derivatives, such as DATETIMEOFFSET for SQL Server)	Alpha, System.String
Double	Implied decimal
Float	Implied decimal
Integer	Decimal, integer
Number	Decimal, implied decimal, integer
Numeric	Decimal, implied decimal

Loading database columns...	...to defined Synergy DBL host variables
Time	Alpha, System.String
Timestamp	Alpha, System.String
Varchar	Alpha, System.String

Note the following:

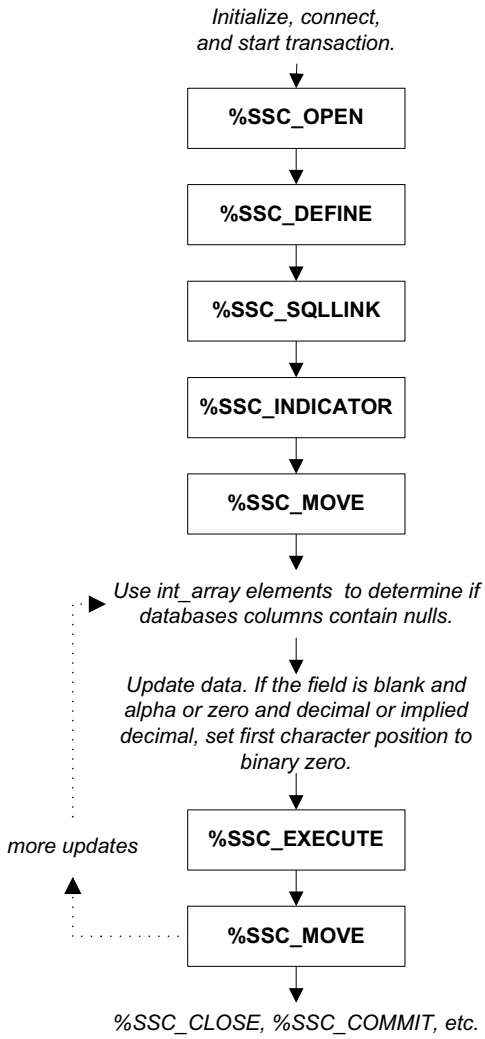
- For alpha host variables, nulls become blanks.
- When using an alpha variable to receive data from a datetime column with microsecond precision, make sure the format string for %SSC_OPTION includes the UUUUUU mask.
- For binary columns, %SSC_EXECIO converts binary zeros to spaces and trims trailing spaces, unless you pass the SSQL_EXBINARY option. (If you use the SSQL_EXBINARY option, %SSC_EXECIO uses the data as is.) %SSC_MOVE, on the other hand, always preserves binary zeros when retrieving data from binary columns.
- For decimal, implied decimal, and integer host variables, nulls become zeros.
- If a Synergy DBL host variable is not large enough to hold the data value assigned to it, the data value will be truncated. The original size of the data value can be determined using %SSC_INDICATOR. For more information, see %SSC_INDICATOR on page 3-33.
- You must use %SSC_INDICATOR to determine if a fetched column was returned with a null value.
- For non-array operations, SQL Connection can move the data in char, date, datetime, varchar, or (when using %SSC_LARGECOL) binary columns to System.String define variables. (System.String define variables are not supported for array-based operations.)

Using %SSC_INDICATOR when updating a column with null

When binding a char, date, datetime, numeric or float column with null, you can use %SSC_INDICATOR to determine if the column is currently null. You can then use this information to determine if the update value should be stored as null.

1. Initialize SQL Connection (%SSC_INIT), connect to the database (%SSC_CONNECT), and start the transaction (%SSC_COMMIT), as illustrated in the diagrams in "Function Call Flow" on page 2-9.
2. Open a cursor with SSQL_SELECT and SSQL_POSITION, and set up the SELECT statement, binding variables as necessary.
3. Define variables.
4. Set up the update statement with %SSC_SQLLINK, binding variables as necessary.
5. Use %SSC_INDICATOR to record null status for retrieved columns.
6. Fetch data with %SSC_MOVE.
7. Execute the update statement with %SSC_EXECUTE (SSQL_STANDARD).

   Test *int_array* elements (returned from the %SSC_INDICATOR call) to determine if database columns contain null values before the update. If a column is null and the bind variable for the column is blank and alpha or zero and decimal or implied decimal, set the first character in the bound field to %CHAR(0), which is binary zero. This instructs the database store the value as null.
8. If there are more rows, fetch data (%SSC_MOVE) and then execute update (%SSC_EXECUTE) in a loop.



9. Rebind and close the cursor and release the connection and commit the transaction as necessary. See "Bulk update" on page 2-13 for information.

Converting dates and times

Date and time columns are defined differently for different databases. For example, in Oracle and Synergy databases, dates have the **date** data type. In SQL Server, dates and times have the **datetime** data type. When you're using SQL Connection functions to *write* to the database, the SQL statements you pass must use the correct formats and commands for the database. Unfortunately, there are no standard commands for this. For example, for Oracle and Synergy databases, you use the TO_DATE() or CAST() functions, and for SQL Server you use the CONVERT() function (unless you're using a **d8** variable with the *YYYYMMDD* format, as discussed below).

On the other hand, when you're using SQL Connection functions to *read* dates and times from the database, these functions pass a date and/or time to your application that's been converted to either an alpha value (if it's passed to an alpha variable) or a Julian date value (if it's passed to a numeric variable).

- If a date or time value is converted to an alpha value, you can use the %SSC_OPTION function to set the date/time mask. This specifies the format for the date or time.
- If a date or time value is converted to a Julian date value, the value is based on the SQL Connection Julian base date. You can get and set the Julian base date with the %SSC_OPTION function. (Note that we don't recommend changing this value.) For information on Synergy DBL routines that handle Julian dates, see the "System-Supplied Subroutines and Functions" chapter of the *Synergy DBL Language Reference Manual*. For more information on the Julian base date and the date/time format mask, see %SSC_OPTION on page 4-8.

What if your application uses **d6** or **d8** variables for dates? Because SQL Connection functions convert returned dates to Julian date values for numeric variables, you won't be able to use **d6** or **d8** variables in SQL Connection functions to directly receive dates unless you want to use the Synergy DBL Julian routines. Otherwise, you'll need to cast these numeric fields as alpha fields. For example, the following sets **date_field** to eight digits of the Julian date value if **date_field** is defined as a **d8**:

```
sts = %SSC_DEFINE(dbchn, cur1, 1, date_field)
```

The following, however, casts the date as an alpha field:

```
sqlp = "SELECT or_odate FROM orders WHERE or_number=3"
If (%ssc_open(dbchn, cur1, sqlp, SSQL_SELECT))
    goto err_exit
sts = %SSC_OPTION(dbchn, SSQL_GETOPT, date_base, date_format, null)
date_format = "DDMMYYYY"
sts = %SSC_OPTION(dbchn, SSQL_SETOPT, date_base, date_format, null)
sts = %SSC_DEFINE(dbchn, cur1, 1, ^A(date_field))
```

In this case, if the retrieved date is February 6, 1958, **date_field** will be set to 06021958. To then write this value back to a database, you could do something like the following for Oracle or SQL Server:

```
sts = %SSC_OPEN(dbchn, cur1, "insert into orders(or_date) where
&               or_number=3 values(to_date(:1, "DDMMYYYY")",
&               SSQL_NONSEL, SSQL_STANDARD, 1, ^A(date_field))
```

When writing to an Oracle or SQL Server database, if your program uses a **d8** variable and the *YYYYMMDD* format for the date, you don't need to use TO_DATE() or CONVERT() to write the date to the database. (The *YYYYMMDD* format is the default for these databases.) You will, however, need to convert the **d8** into an alpha with the ^A() function.

For a date conversion example, see **exam_create_table.dbl** in the connect\synsqlx subdirectory of the main Synergy/DE installation directory.

Numeric database columns

To maximize the portability of your code to various databases, we recommend using the numeric type for columns when writing CREATE TABLE statements. Creating a database column as numeric will ensure that the column will map to a database data type suitable for commercial data storage equivalent to at least a **d28.10**. Some databases also allow integer storage. SQL Connection will translate between the database numeric data types and the Synergy DBL variables, whether they are integer, decimal, or implied decimal.

Updates and Locking

This section presents general information on concepts, features, and procedures that differ from one database management system to another. We include this information as a starting point; for complete information on these subjects, see your database documentation.

When multiple users access and modify data from the same database, there can be data access conflicts (multiple users attempting to access the same data) and update conflicts (different versions of the same data being modified and committed to the database at the same time). This is possible because users of front-end applications can simultaneously access the same data on the same database. Additionally, when a user accesses data for viewing or modifying, one or more rows of data are copied from the database into network buffers or into host variables in the front-end application. From this point on, or until the data is stored back into the database, the data the user views may no longer be the same as the data residing in the database. Other users may have accessed and modified the data between the time the user accessed data and the time the data was committed.

To prevent conflicts and ensure data integrity, relational database management systems (RDBMSs) provide controls that enable developers to specify how data will be accessed. Row locking is one such method that's commonly employed, but it differs from one database management system to another. And in some situations, the front-end application must be able to verify that updates made to the database by one user do not overwrite updates made by another user (see "Optimistic locking and unique row identifiers" on page 2-44).

Row locking

To balance the need for good performance against the potential for data conflicts when multiple users access a database, two types of locking are generally employed by database engines: pessimistic locking and optimistic locking. In simple terms,

- *pessimistic locking* assumes that multiple users might be accessing the same data and attempts to prevent conflicts. Pessimistic locking locks data for much of the duration of a transaction. For SQL Connection (when the correct commands are used) pessimistic locking locks rows from the first fetch with %SSC_MOVE or insert until the transaction ends with %SSC_COMMIT, and an error is returned if another user attempts to access locked data. (See figure 2-4 below.)
- *optimistic locking* assumes that there will *not* be a conflict for data resources and therefore relies on the front-end application to ensure data integrity. When optimistic locking is used and data is committed by a user, the database engine does not check to see if data has been accessed by another user, and you may get lock failures at write time rather than read time.

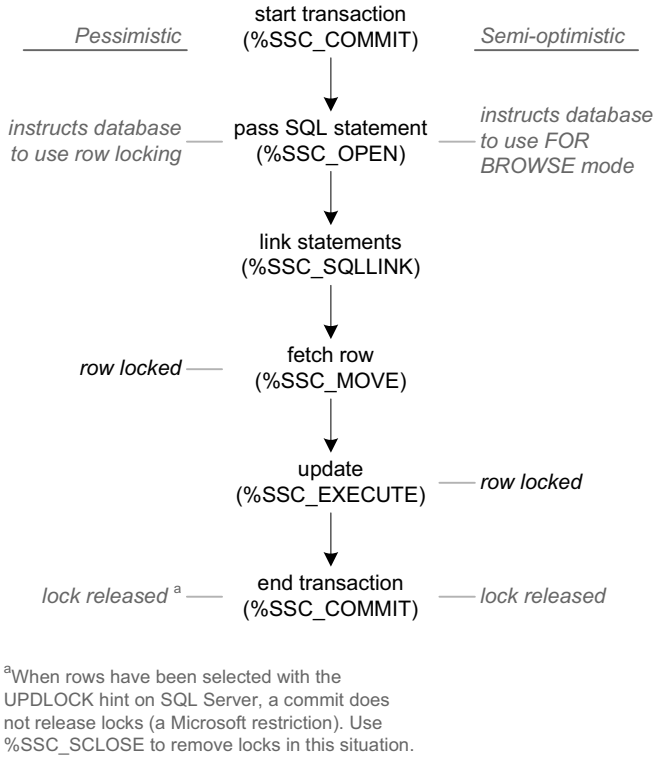


Figure 2-4. Pessimistic versus semi-optimistic locking.

Different databases management systems use different methods to initiate locking. For many that follow ANSI standards, a SELECT FOR UPDATE operation invokes the database engine's inherent locking method (which typically is pessimistic). For example, Oracle uses pessimistic locking when a SELECT FOR UPDATE operation is performed. On the other hand, when using SQL Server with the VTX12_ODBC or VTX12_SQLNATIVE database driver, pessimistic locking is generally invoked automatically on SELECT if you use a dynamic cursor (the default for VTX12_ODBC and VTX12_SQLNATIVE) with the UPDLOCK hint and, optionally, the ROWLOCK hint. Otherwise, SQL Server does not use row locking with these database drivers. Also note that when rows have been selected with the UPDLOCK hint, a commit does not release locks.

Allowing the database engine to use pessimistic locking may result in longer locks on data and increased demand on database resources, especially when many users access a database simultaneously.

- As an alternative, you can use unique row identifier information to optimize transactions when using pessimistic locking. This enables SQL Connection to use the unique row identifier information retrieved with the fetch to locate the fetched rows on update (instead of having to go through the index). See "Optimistic locking and unique row identifiers" below for more information.
- For SQL Server, we recommend using a select cursor and an update cursor with a primary key constraint rather than using FOR UPDATE OF and %SSC_SQLLINK (positioned update mode). Using positioned update mode is about 25% slower.

Optimistic locking and unique row identifiers

The concept of unique row identifiers is central to relational database operation. For row locking to occur at all, the database engine must be able to identify each row with a unique identifier, or pointer. For example, SQL Server enables you to include a rowversion (timestamp) column. In an SQL database table, the unique identifier is usually derived from some combination of the row location within the database and a unique numeric value or timestamp.

For optimistic locking, you can compare these values in your SQL Connection program to determine if a row has been updated by another user or process. You can then write your program to handle both successful and failed comparisons. Verifying unique row identifiers

- provides an additional safeguard for data integrity.
- enables you to write an application that locks data resources more efficiently with lower overhead than would be achieved by depending on the database's inherent locking mechanism. This increased efficiency is achieved because with unique row identifiers you can lock resources on a row-by-row basis, affecting only those rows where the unique identifier hasn't changed and locking them only for the duration of the update transaction. Using unique row identifiers can reduce contention for data in a multi-user application and thereby lead to increased throughput for all other application requests.
- improves performance for positioned updates on some databases, such as Oracle. Because SQL Connection can use the unique row identifier for an update, the database doesn't have to use the index a second time to locate the fetched row via a WHERE clause.

Note that SQL Connection has a convenient method for using unique row identifier information as a condition for update. See "Using SQL Connection's automatic verification" on page 2-48 for information.

MySQL and optimistic locking

To verify a unique row identifier for a MySQL database, use the method described in "Using SQL Connection's automatic verification" on page 2-48 or use a CURRENT_TIMESTAMP clause with a timestamp column in an ON UPDATE statement. For an example of the latter, see the **exam_create_table.dbl** sample program included in the Connectivity Series distribution.

SQL Server and optimistic locking

SQL Server provides the following methods for client-side concurrency control:

- Cursors, as defined by SQL Server, enable applications to perform a positional fetch within result sets and update rows. The client and server data-access components work in unison to manage concurrent data access and consistency. See %SSC_CMD on page 3-7 and SQL Server documentation for more information.
- At a lower level, a rowversion pseudo column enables you to manually control data concurrency. See "Using a rowversion column" below.
- Globally unique identifiers (GUIDs) also enable you to manually control data concurrency. See "Using GUID columns" below.

You can use the method described in "Using SQL Connection's automatic verification" on page 2-48, but for performance reasons, it's not recommended for SQL Server.

Using a rowversion column

With SQL Server, you can use a rowversion column to ensure data integrity. The rowversion column has a user-defined **varbinary(8)** data type and is updated with the current date and time when an INSERT or UPDATE command is executed. When creating a table, you can optionally specify a rowversion column, but note that a table can have only one such column and that rowversion columns are accessible to client applications only as read-only columns. (In addition to identifying the row, these IDs also identify a version of the row—i.e., the state of a given row at a given time. If you change the contents of a row, the row's ID column will get a new value. Think of it as an RFA, a record file address, that changes every time a column in the row is updated.)

Rowversion columns are particularly useful for synchronizing multiple remote databases that are replications of a central database. For example, when data in several remote databases is committed to a large central database on a periodic basis, the rowversion column can be used to verify that modified data is not overwritten with earlier data.

Note the following:

- In SQL statements, you must enter the column name as rowversion or timestamp in lowercase letters.
- If you do not specify the rowversion column when creating a table in a SQL Server database, you will get an error with SQL Connection when using SSQL_FORUPDATE.

See SQL Server documentation for more information.

Using GUID columns

You can use the NEWID() Transact-SQL function to create a globally unique identifier value (GUID) for a row. To produce a unique identifier value in an inserted row, either the table must have a DEFAULT clause specifying the NEWID() function, or this function must be included in the INSERT statement (which is not necessary for a rowversion column). However, unlike a rowversion column, you can fetch GUID column values with a SELECT statement. (Rowversion columns are invisible to SELECT queries.)

The following example demonstrates how to create a unique identifier both automatically and manually. The first INSERT statement automatically creates a NEWID() value for the GUID column (triggered by the DEFAULT NEWID() clause in the CREATE TABLE command). The second INSERT manually generates the value.

```
CREATE TABLE e_anniv (
      guid UNIQUEIDENTIFIER CONSTRAINT Guid_Default DEFAULT NEWID(),
      Start_Date DATETIME,
      Employee_Name VARCHAR(60),
      )
GO
INSERT INTO e_anniv (Start_Date, Employee_Name) VALUES ('7/1/1976','John')
INSERT INTO e_anniv VALUES (NEWID(), '3/8/1982','Mary')
GO
```

Note that you should use an **a16** to store GUIDs in your SQL Connection program, and if you use a GUID in a stored procedure, you must use SSQL_EXBINARY.

Oracle Server and optimistic locking

For Oracle, you cannot write client-side concurrency code for optimistic locking unless you are using Oracle 10g or later with the ORA_ROWSCN column when the table is set up with ROWDEPENDENCIES enabled. In this case, you can retrieve the SCN in a binary field and use it in a WHERE clause.

You can, however, use ROWID values to identify the current row during updates and deletes. ROWID has the format *BBBBBBBB.RRRR.FFFF* (hexadecimal), where *BBBBBBBB* is the block in the database file, *RRRR* is the row in the block (0 = first row), and *FFFF* is the database file. For example, 0000000E.000A.0007 denotes the 11th row in the 15th block in the 7th file.

Note the following:

- Adding a FOR UPDATE clause locks fetched rows.
- If you want to use COMMITs and FETCHs with two different cursors, do *not* use the CURRENT OF clause. CURRENT OF links cursors, so the COMMIT statement will close the operation and make good all updates. You will also lose the selected result set. Instead, select the ROWID and use that value to identify the current row during the update or delete.

Oracle Rdb and optimistic locking

Oracle Rdb uses DBKEY as a unique row identifier. DBKEY values are binary values. When you access a row by DBKEY, the database system can retrieve, delete, or update that row directly, without accessing an index or sequentially scanning a table row by row.

By default, DBKEY values are guaranteed to be valid until you end the transaction in which you retrieve them. However, DBKEY values will remain valid until you detach from the database if you include the clause DBKEY SCOPE IS ATTACH when you declare the schema for the database to which the DBKEY values belong.

To specify a DBKEY as a value expression, use the keyword DBKEY. This keyword is valid only in a selection list (to retrieve DBKEY values) or in a basic predicate with the equal (=) operator (to access rows by the DBKEY values your program retrieves). For example, your program might contain the following types of statements for accessing an Oracle Rdb database:

```
SELECT col_1, col_2, DBKEY INTO col_1_parm, col_2_parm, dbkey_parm
      FROM table_a WHERE col_1 = input_retrieval_parm
UPDATE table_a SET col_2 = update_parm WHERE DBKEY = dbkey_parm
```

Even though DBKEY values are stored in binary format in the schema, you must declare the host variable into which your program stores a DBKEY as a character string. The size of the character string depends on which operating system Oracle Rdb is running. Check Oracle documentation for the required length of the character string for each operating system.

Informix and optimistic locking

With Informix, the ROWID keyword can be used in RDSQL statements to refer to the C-ISAM record number associated with a row in a database table. ROWID can be thought of as a hidden column in every table. SELECT statements will not return ROWID values unless you specify ROWID in the column list. The following example returns the ROWID value for each row:

```
SELECT ROWID, * FROM table
```

The next statement, however, does not return ROWID:

```
SELECT * FROM table
```

ROWID can also be used in WHERE clauses to select rows based on their C-ISAM record number. This feature is especially useful when there are no other unique columns in a table. For example:

```
SELECT ROWID, * FROM table WHERE ROWID > 7
```

Note that if a row is deleted from the table, its ROWID may be assigned to a new row.

Using SQL Connection's automatic verification

SQL Connection has a convenient method for using unique row identifier information as a condition for update. This method frees you from having to code row identifier comparisons in your SQL Connection program. With this method, the unique row identifier information is saved when the row is fetched, and then it is automatically used as a condition for the update. You can think of it as an under-the-hood WHERE clause that compares the fetched row identifier with the row identifier that is there at the update—something like "WHERE current_row_id = fetched_row_id".

1. Use a SELECT FOR UPDATE statement and specify the SSQL_FORUPDATE and SSQL_POSITION options in your call to %SSC_OPEN.
2. Use %SSC_SQLLINK in conjunction with %SSC_EXECUTE.

Note that for performance reasons we don't recommend using SELECT FOR UPDATE statements with SQL Server (see note in "Row locking" on page 2-42). However, if you do use this method, note that you must have both a rowversion (timestamp) column (see "Using a rowversion column" on page 2-45) and a unique index.

Transactions and Autocommit

When a database engine is configured with autocommit on, all SQL operations are committed as soon as they are executed. For better performance and transaction control, however, we recommend that you turn autocommit off and use the SQL Connection functions %SSC_COMMIT and %SSC_ROLLBACK to control transaction blocks, as in the following typical process:

```
begin transaction (%SSC_COMMIT)
update order data
if error on update
  rollback (%SSC_ROLLBACK)
else
  update customer
  if error on update customer
    rollback (%SSC_ROLLBACK)
  else
    commit (%SSC_COMMIT)
  endif
endif
```

Unless you use autocommit, all transactions must either be committed with %SSC_COMMIT or rolled back with %SSC_ROLLBACK. The default is database-dependent, but for all databases %SSC_COMMIT or %SSC_ROLLBACK must be used after data access (DML) operations, and you should call %SSC_COMMIT before invoking a DML operation (this includes insert, update, delete, and select commands). Note, however, that the database determines if %SSC_COMMIT must be called before the database is actually modified. (See %SSC_COMMIT on page 3-17 and %SSC_ROLLBACK on page 3-52 for more information.)

If a database engine employs a cursor caching mechanism, it is critical to explicitly end transactions. If you do not, the cache will quickly exhaust system resources.

If you are accessing an Oracle database, Oracle recommends that you explicitly end every transaction in your application with a COMMIT or ROLLBACK statement. This includes the final transaction, the one before you disconnect. If the application terminates abnormally and you have not done this, the last uncommitted transaction is automatically rolled back.

Row locking and transactions

While write transactions or read-with-lock transactions are in process, affected data rows are locked by the database engine. Depending on the type of lock, adjacent rows may be locked as well (for example, in page-level locking). A lock persists until the database engine commits or rolls back the data, thereby closing the transaction and releasing any affected rows. However, note the following:

- If autocommit is off and you don't commit the transaction using %SSC_COMMIT, SQL Connection will automatically roll back any pending transactions when the %SSC_RELEASE function is called. For more information on locking, see "Updates and Locking" on page 2-42.
- A commit does not cause VTX12_ODBC or VTX12_SQLNATIVE to automatically release locks for rows selected with an UPDLOCK hint. (This is a Microsoft restriction.) If the application won't read another row, be sure to use %SSC_SCLOSE to remove such locks. Additionally, use %SSC_SCLOSE before a rebind if you do not immediately close the cursor with %SSC_CLOSE.

Stored Procedures

A stored procedure is a pre-compiled, ready-to-execute command stored in a database and managed as a database object. Stored procedures run as stand-alone programs on the database server and are invoked by internal or external requests. Internal requests originate within the database and are invoked by other stored procedures, triggers, and agents. External requests originate from remote client applications, local server applications, and other database systems.

One of the primary advantages of stored procedures is that they enable you to move application code and business logic to the server. There are other advantages, and there are some disadvantages.

Advantages:

- *Less redundancy.* A stored procedure is available to all applications that access a database. This enables you to store business logic in one place—the database—rather than in each application.
- *Consistency.* A stored procedure provides a single process for all users, eliminating inconsistencies that are likely to occur when logic is coded separately in each application that accesses the database.
- *Maintainability.* Stored procedures recognize changes to database schemas. Although stored procedures are pre-compiled, stored procedures verify column definitions when they are run. If there are changes, these changes and any necessary data type conversions are made at runtime.
- *Manageable, well-defined logic modules*. Stored procedures are modified independently of the application source code and perform a single task.
- *Faster execution.* It's often faster to run stored procedures than to run lengthy or repetitive SQL operations. See related disadvantage below.
- *Reduced network traffic*. An operation that would otherwise send hundreds of lines of SQL code over a network can be made into a stored procedure, which requires the network to handle only one statement: the statement that invokes the stored procedure.
- *Security.* A user can be given permission to execute a stored procedure even if the user doesn't have permission to execute the procedure's statements directly. You can create a very secure and extensible environment by creating applications that use only stored procedures.
- *Additional prefetch information*. With a single database call, client applications can retrieve the result of an operation as well as the number of rows in the result set.

Disadvantages:

- *Lack of portability.* Stored procedures are not portable from one brand of database to another. For example, SQL Server and Sybase cannot run stored procedures created for Oracle, and Oracle cannot run stored procedures created for SQL Server or Sybase.
- *Potential for reduced performance*. Overburdening the server with stored procedure processing, in addition to standard RDBMS tasks, may degrade database performance.

- *Difficult debugging*. Nested stored procedures (stored procedures called by other stored procedures) are difficult to debug. Stored procedures invoked from event-driven triggers are even more difficult to debug.
- *Reduced stability.* If a stored procedure that was installed as an external DLL and run within the address space of the database engine fails, the server may also fail.

Invoking stored procedures

Use one of the following methods to invoke a stored procedure in an SQL Connection program:

- To fetch data from a SQL Server result set, use the %SSC_OPEN / %SSC_MOVE method illustrated in the example programs **stp_sqlsrv1.dbl** and **stp_sqlsrv2.dbl**. Note that this method works only with SQL Server, and you must include the EXECUTE command in the SQL statement that invokes the stored procedure.
- If the stored procedure requires parameters, use %SSC_EXECIO. This works with any database supported by SQL Connection, but it cannot be used to retrieve data from a SQL Server result set. For SQL Server, you do not need to include the EXECUTE command. You can just pass the name of the stored procedure followed by parameters. See the example programs **stp_ora.dbl**, **stp_odbc.dbl**, and **stp_sqlsrv.dbl**.
- For other cases, use %SSC_EXECIO or %SSC_EXECUTE. These work with any database supported by SQL Connection.

For an illustration of the function call flow for the latter two methods, see "Stored Procedure" on page 2-15. For information on the example programs, see "Writing an SQL Connection Program" on page 2-2.

Notes on Oracle stored procedures

For Oracle databases, stored procedures are called *subprograms*. There are two types of Oracle subprogram, both of which are written in PL/SQL, a procedural language extension of SQL. These two types are procedures and functions. Procedures and functions are similar. Both are typically written to perform a single task, but functions return a value, so you can use them within SQL expressions. This includes WHERE clauses in SQL statements and control structures within PL/SQL. However, to use a procedure, you must pass the procedure by name (in an %SSC_OPEN call), and you must use the %SSC_EXECIO function. For an example, see **stp_ora.dbl**, a file included in the Connectivity Series distribution.

In essence, subprograms are named PL/SQL blocks that have been compiled into p-code and stored in an Oracle database. Once the p-code is in the database, it is ready to run. Subprograms may take and return user-supplied parameters, and any application connected to a database can access the database's subprograms by name. When an application accesses a subprogram, the subprogram is passed to the PL/SQL engine, which maintains a single copy of the subprogram for all applications to use.

Subprograms are created and modified with the CREATE OR REPLACE PROCEDURE statement. Packages are created and modified with the CREATE OR REPLACE PACKAGE statement.

Refer to Oracle documentation to learn more about creating packages and writing PL/SQL subprograms.

For information on invoking stored procedures, see "Invoking stored procedures" on page 2-52.

Using packages to group subprograms

Related Oracle subprograms can be grouped into packages. Packages are named PL/SQL modules that provide a convenient method for grouping logically related components (types, items, and subprograms). Packages also enable you to create public and private components. Public components can be called from and shared with internal and external callers. Private components are available only to components within the same package.

If a subprogram is part of a package, the PL/SQL engine loads the entire package the first time the subprogram is used. Thereafter, calls to any of the components in the package are processed immediately and without additional overhead. Public variables and cursors persist for the duration of a session and remain unaffected by transactions.

Notes on SQL Server stored procedures

SQL Server refers to its version of SQL as Transact-SQL (T-SQL). T-SQL includes not only standard SQL, but also procedural language extensions that enable you to create user-defined stored procedures. In SQL Server, stored procedures are named T-SQL blocks that may take and return user-supplied parameters. Stored procedures are parsed, optimized, and then saved in the database. When a stored procedure is called, the T-SQL processor loads the procedure, runs the procedure, and then retains the executable image. Subsequent calls to the stored procedure use this cached in-memory version, reducing system overhead and improving performance.

In addition to user-defined stored procedures, SQL Server includes two other types of stored procedure: system stored procedures and extended stored procedures. System stored procedures are used to perform many administrative functions. These procedures are created and stored in the master database and their names begin with the **sp_** prefix. Extended stored procedures enable you to create external routines in programming languages that enable you to create dynamic-link libraries (DLLs). Extended stored procedures are run and appear to end-users as user-defined stored procedures.

SQL Server stored procedures are created with the T-SQL CREATE PROCEDURE statement and modified with the ALTER PROCEDURE statement. See Microsoft documentation for information on T-SQL and writing stored procedures.

For information on invoking stored procedures, see "Invoking stored procedures" on page 2-52.

Optimization

To illustrate the importance of optimizing your SQL code, imagine the following scenario: Your Synergy data processing application runs in an office with approximately 100 simultaneous users. Each user requires 50 to 100 open cursors at any given moment, and each cursor requires about 10K for memory allocation. For cursors alone, this could quickly absorb up to 100 MB in database resources. To handle this demand as efficiently as possible, SQL Connection enables you to optimize your SQL Connection application in several ways, such as reusing cursors.

This section introduces some of the issues that affect performance. Refer to your database documentation for additional information.

Reusing cursors

For optimal performance, it may be best to reuse a cursor if you're going to redo the same operation a little later in your program. Cursor reuse can significantly improve database and network performance. It saves time opening the cursor, and it uses less memory. See "Reusing cursors" on page 2-28.

Optimizing data transfer

One advantage of relational databases, especially in client/server situations, is that you can retrieve only those fields (or columns) that you intend to update. In other words, you may not need to transfer all columns in a row, unlike with ISAM files for which the entire record is retrieved. If you take advantage of this capability, you can potentially cut down on a great deal of network traffic.

To optimize data retrieval across a network, we suggest that you consider using subroutines to access data from the database and that you have multiple versions depending on the job being performed. The goal is to transfer the least amount of data possible.

Creating tables

Table creation is the slowest operation for a relational database. Using a database table as a work file (or "scratch" space) will lead to poor performance. Keep in mind that it is seldom necessary to maintain transaction integrity on the work files.

Instead of creating a relational database table as a scratch space, you should consider using a standard Synergy ISAM file for this purpose. ISAM files will always outperform a relational database in these situations. In fact, many database consultants recommend using a mixture of ISAM and relational files for high performance online transaction processing (OLTP). If you must use a database table, consider leaving work tables created, with the data removed, instead of deleting and recreating them.

Optimizing atomic operations with %SSC_SQLLINK

Two cursors are often used for data access and updates. The first cursor is used to fetch the data from the database, and the second cursor is used to update the data for each row fetched. (In this mode of operation, autocommit is usually used.) To reduce resource requirements and overhead for an atomic operation, SQL Connection allows positioned update mode, which enables you to update a single row using the same cursor you fetched the row with. In other words, the same cursor that's opened for the %SSC_MOVE function can be used to commit data with the %SSC_SQLLINK / %SSC_EXECUTE combination. See %SSC_SQLLINK on page 3-56, and for an example, see **exam_fetch_update.dbl**, which is in the connect\synsqlx subdirectory of the main Synergy/DE installation directory.

Improving network performance with prefetch caching

To improve network performance and reduce database operations, SQL Connection uses prefetch caching for fetches made with %SSC_MOVE (but not %SSC_EXECIO) if the cursor is opened with SSQL_SELECT and SSQL_STANDARD and if none of the %SSC_OPEN options that disable caching are passed.

If SQL Connection is able to use prefetch caching (and the cache size is optimal), only one network packet transfer is necessary to retrieve the result set for an SQL statement. However, if prefetch caching is not used, network packets are sent in both directions for every operation needed for the SQL statement. SQL Connection uses prefetch caching for both direct and SQL OpenNet connections.

The default prefetch buffer size is 32,768 bytes, but you can change this by passing a different value for the *bufsize* argument for %SSC_INIT. See the %SSC_INIT Discussion on page 3-35.

Note that some fetch/open combinations may require long searches for more data to fill the prefetch buffer when there is no more data to be returned. If this is the case, consider using SSQL_ONECOL or SSQL_POSITION (%SSC_OPEN options) to prevent prefetch caching.

Selecting the optimal cursor type

If you use the VTX12_ODBC, VTX12_SQLNATIVE, or VTX11 database driver, you may be able to improve performance by changing the type of cursor that your application uses. The %SSC_OPEN and %SSC_CMD functions have several options that enable you to do this. For example, if your application reads large amounts of data from a SQL Server database and you use VTX12_ODBC or VTX12_SQLNATIVE, you can probably improve performance by setting the cursor type to forward-only (SSQL_CURSOR_FORWARD_ONLY) and read-only (SSQL_RO_CURSOR). For more information, see "Cursor types" on page 2-29.

Reducing the number of sockets used for SQL Server

When connecting to SQL Server, you can use the SQL Server shared memory protocol, which can greatly improve performance by reducing the number of TCP/IP sockets used for a connection. See "Using the SQL Server shared memory protocol" on page 2-26.

Reducing memory and enabling more concurrent users

There are a couple of ways to reduce the memory used by an SQL Connection application, which may enable it to support more concurrent users.

The first method is to lower the *maxcur* and *maxcol* values passed to %SSC_INIT. This works on all platforms. For more information, see %SSC_INIT on page 3-35.

The second method is to use the **-s** option for **vtxnetd** running on a Windows server. Reducing the **-s** setting reduces the size of the thread stack allocated to **vtxnetd**, which lowers the amount of memory used for SQL OpenNet. For more information, see "The vtxnetd and vtxnet2 Programs" in the "Configuring Connectivity Series" chapter of the *Installation Configuration Guide*.

Optimizing queries

A query's construction can greatly affect an SQL Connection program's performance. Below are some suggestions, but you should also refer to some general SQL reference works and your database documentation.

- Be sure to include either table names or pseudonyms when defining columns in a SELECT statement, especially when there is a join. For example:

  ```
  SELECT a.name,b.address FROM accounts a, addresses b WHERE
         b.account_id = a.id
  ```

  This relieves the query optimizer from having to determine where a column has come from and from checking for conflicts (i.e., columns with the same name in other tables). Additionally, if someone later adds a column with the same name as a column used in your query, it won't cause a conflict, which would prevent the query from working.

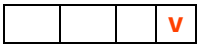
- Use an ORDER BY clause if you expect a query to return more than one column. Otherwise, data may be returned in seemingly random order, especially if there is no primary key constraint.
- Do not use a function in a WHERE clause. This can cause very poor performance because it prevents the query optimizer from using an index, resulting in a full table scan. Generally, you can eliminate the need for functions in WHERE clauses by optimizing data storage. For example, if a field is used only in uppercase form, store it that way. Or, if you must allow mixed case, consider creating a second column with the data in upper case and lower case just for searching.
- Never use a null in a WHERE clause. Such clauses (e.g., "WHERE column_name=null") are often the result of columns that default to null. It's better to give the column a different default value, one that's meaningless in the context. For example for a column that defines a customer credit limit, you could use -1 as the default.
- Review the query plan to make sure your query is optimized. See your database documentation for information.

3

Database Functions

SQL Connection database functions are directly related to SQL-based operations and data access. Each database function returns a value and can be used any place a literal can be used in a Synergy program.

%INIT_SSQL – Initialize SQL Connection



value = %INIT_SSQL

Return value

value

This function always returns 0. (**i**)

Arguments

None.

Discussion

%INIT_SSQL initializes SQL Connection on OpenVMS and allocates necessary memory. (On Windows and UNIX systems, set system option 48 to initialize the SQL Connection system and to instruct the runtime to allocate the necessary memory.) For more information, see "Installing, Configuring, and Initializing" on page 1-6.

When you're using SQL Connection on OpenVMS, you must make %INIT_SSQL the first function call.

%SSC_BIND – Bind host variables for non-SELECT statement

WT	WN	U	V

value = %SSC_BIND(*dbchannel, dbcursor, numvars, var[, ...]*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

SSQL_FAILURE Failure

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

dbcursor

The logical cursor number within the range from 1 through the maximum number specified by *maxcur* during %SSC_INIT. The cursor must have been opened by %SSC_OPEN. (**n**)

numvars

The number of bind variables in the non-SELECT statement, up to the maximum number of columns specified by the *maxcol* argument in %SSC_INIT. If *numvars* is negative, the bind variables are overwritten rather than appended. If *numvars* is positive, the bind variables are appended to the variables for the non-SELECT statement. (**n**)

var

Host variable(s) to be bound to the non-SELECT statement. The number of *var* variables passed must equal the value of *numvars*. (**a**, **n**, or String)

Discussion

%SSC_BIND binds host variables to variables for a non-SELECT statement. It affects only variables that are used when %SSC_EXECUTE is called. See "Binding data" on page 2-34 for more information on binding, and note the following:

- The total number of bind variables used by %SSC_EXECUTE must match the number of columns defined by the non-SELECT statement for an open cursor. For example, if there are 20 host variables in an UPDATE statement, you could use one %SSC_BIND call to append definitions for all 20 variables, or two %SSC_BIND calls with 10 variables each, or any other combination, as long as the number of variables defined by %SSC_BIND *exactly matches* the number of variables described by %SSC_OPEN in the non-SELECT statement before %SSC_EXECUTE is called.
- The String data type (System.String) is not supported for *var* for array-based operations.
- If you use ^VARARGARRAY, note that *numvars* is the last declared argument for this routine.

Examples

The following example shows how to bind two Synergy DBL variables associated with *cur1*.

```
record order_rec
       ord_num            ,d6
       ord_cust           ,d6
.
.
.

sqlp = "UPDATE orders SET or_number = :1 WHERE or_customer = :2
if (%ssc_open(dbchn, cur1, sqlp, SSQL_NONSEL))
  goto err_exit
if (%ssc_bind(dbchn, cur1, 2, ord_num, ord_cust))
  goto err_exit
if (%ssc_execute(dbchn, cur1, 1))
  goto err_exit
```

%SSC_CANCEL – Cancel outstanding requests

WT	WN	U	V

value = %SSC_CANCEL(*dbchannel*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

Discussion

%SSC_CANCEL cancels outstanding database requests when a user cancels a database operation. The results of the function depend on the database, and some databases, such as Oracle Rdb, don't support this type of call. For these databases, nothing is canceled; the function simply returns SSQL_NORMAL. Databases that do support a cancel, however, may do the following: cancel any outstanding database requests, cancel execution of current SQL statements (or the entire set of currently processing batch commands), and flush and pending results.

Use this function only in program-exit routines that are called when the user cancels a database operation—for example, in a routine that's called after CTRL+C is trapped or in a close method for a Windows application.

%SSC_CLOSE – Hard close one or more open cursors

WT	WN	U	V

value = %SSC_CLOSE(*dbchannel, dbcursor[, ...]*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

dbcursor

One or more logical cursor numbers within the range from 1 through the maximum number specified by *maxcur* during %SSC_INIT. (**n**)

Discussion

%SSC_CLOSE hard closes one or more logical cursors opened with %SSC_OPEN. A hard close closes the cursor on the target database. (For information on soft closes, see "Cursors" on page 2-27 and %SSC_SCLOSE on page 3-54.)

The cursor(s) specified with *dbcursor* must have been opened by %SSC_OPEN. Once you close a logical cursor, that cursor is no longer valid. (Note that if *dbcursor* is not an open cursor, you will get a return value of SSQL_NORMAL, not SSQL_FAILURE.)

The *dbcursor* argument is cleared to 0 when a cursor is hard closed.

Note the following:

- A commit or rollback transaction may or may not close open cursors automatically. You should always explicitly close cursors before a commit or rollback.
- A table can be dropped (that is, deleted or removed) only if all cursors are hard closed.

If you use ^VARARGARRAY, note that *dbcursor* is the last declared argument for this routine.

Examples

For an example that uses %SSC_CLOSE, see %SSC_COMMIT on page 3-17.

%SSC_CMD – Set database-specific options

WT	WN	U	V

value = %SSC_CMD(*dbchannel, [dbcursor], option, parstring*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

SSQL_FAILURE Failure

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

dbcursor

(optional) This argument is currently ignored. (**n**)

option

An option that executes a database-specific command. (**n**)

parstring

Parameters for the specified option. Parameters must be separated by spaces. The maximum size of the string is 60 characters. (**a**)

Discussion

%SSC_CMD executes a database-specific command (*option*).

Supported *option* values are listed in the "Option" column in the table below and are defined in the **ssql.def** file distributed with Connectivity Series. Note the following:

- If the *option* value is not supported by the database driver, it is ignored.
- *Option* settings whose duration of setting is listed as "Connection" in the table below last for the duration of the connection or until a subsequent %SSC_CMD call changes the setting.
- Many of these options affect subsequent opens with %SSC_OPEN, but don't affect current cursors.

Database-Specific Options			
Option	**Description/parameters**	**Database driver(s)**	**Duration of setting**
SSQL_CACHE_CHAIN	Instructs %SSC_RELEASE to cache connections and preserves the cache when chaining to other programs. **Parameters:** none See "SSQL_CACHE_CHAIN" on page 3-11.	All (on Windows and UNIX only)	Runtime instance
SSQL_CACHE_CONNECTION	Instructs %SSC_RELEASE to cache connections, but allows cached connections to be closed when chaining to other programs. **Parameters:** none See "SSQL_CACHE_CONNECTION" on page 3-12.	All (on Windows and UNIX only)	Runtime instance
SSQL_CMD_SCROLL	For a scrolling cursor, determines which result set row will be retrieved with the next fetch. **Parameters:** SSQL_SCROLL_CURRENT SSQL_SCROLL_FIRST SSQL_SCROLL_LAST SSQL_SCROLL_NEXT (default) SSQL_SCROLL_PRIOR SSQL_SCROLL_ABSOLUTE SSQL_SCROLL_RELATIVE See "SSQL_CMD_SCROLL" on page 3-12.	VTX0 VTX11 VTX12_ODBC VTX12_SQLNATIVE	Cursor
SSQL_CURSOR_TYPE	Sets an ODBC cursor type for the next %SSC_OPEN. **Parameters:** SSQL_CURSOR_DYNAMIC SSQL_CURSOR_FORWARD_ONLY SSQL_CURSOR_KEYSET_DRIVEN SSQL_CURSOR_STATIC The default for the VTX12_* drivers is SSQL_CURSOR_DYNAMIC. The default for VTX11 is SSQL_CURSOR_FORWARD_ONLY. See "SSQL_CURSOR_TYPE" on page 3-13.	VTX11 VTX12_ODBC VTX12_SQLNATIVE	Connection

Database-Specific Options (Continued)			
Option	**Description/parameters**	**Database driver(s)**	**Duration of setting**
SSQL_KEEP_OPEN	Prevents runtime from closing connections on a program chain. **Parameters:** none See "SSQL_KEEP_OPEN" on page 3-13.	All (on Windows and UNIX only)	Runtime instance
SSQL_LANGVER	Specifies Oracle parser syntax compatibility. **Parameters:** OCI_NTV_SYNTAX OCI_V7_SYNTAX (default) OCI_V8_SYNTAX See "SSQL_LANGVER" on page 3-13.	VTX0_*n*	Connection
SSQL_NEW_BLOBS	Specifies use of BLOB/CLOB instead of LONG RAW/LONG. **Parameters:** yes|no Default is no. See "SSQL_NEW_BLOBS" on page 3-14.	VTX0 (when connected to Oracle 9, Oracle 10, or Oracle 11)	Connection
SSQL_ODBC_AUTOCOMMIT	Turns autocommit on or off. Yes turns autocommit on, which means that every SQL statement is automatically committed. **Parameters:** yes|no Default is no for SQL Server and MySQL, yes for ODBC and Synergy Database.	VTX11 VTX4 VTX12_ODBC VTX12_SQLNATIVE VTX14	Connection
SSQL_OLD_ZONEDDATE	Specifies pre-7.1 behavior if %SSC_MOVE is used to move date fields to zoned fields. **Parameters:** none See "SSQL_OLD_ZONEDDATE" on page 3-14.	All	Connection
SSQL_ONEPID	Ensures that a single process is used for a transaction. **Parameters:** yes|no Default is no. See "SSQL_ONEPID" on page 3-14.	VTX2	Connection

Database-Specific Options (Continued)			
Option	**Description/parameters**	**Database driver(s)**	**Duration of setting**
SSQL_RAWDATE	Specifies whether to return date/time untouched. **Parameters:** yes|no Default is no. See "SSQL_RAWDATE" on page 3-14.	All	Connection
SSQL_RETURN_ROWID	Determines whether a row ID will be returned for each SQL statement. **Parameters:** yes|no Default is yes. See "SSQL_RETURN_ROWID" on page 3-14.	VTX0 VTX5	Connection
SSQL_RO_CURSOR	Specifies fast-forward cursors (SQL_CO_FFO) if possible for next cursor opened with %SSC_OPEN. **Parameters:** yes|no Default is no. See "SSQL_RO_CURSOR" on page 3-14.	VTX12_ODBC VTX12_SQLNATIVE	Connection
SSQL_SQL_BULK_INSERT	Enables or disables bulk inserts. **Parameters:** yes|no Default is no. See "SSQL_SQL_BULK_INSERT" on page 3-15.	VTX12_ODBC VTX12_SQLNATIVE	Connection
SSQL_SYB_BLANK	Specifies whether to return blank instead of null. **Parameters:** yes|no Default is no. See "SSQL_SYB_BLANK" on page 3-15.	VTX2	Connection
SSQL_TIMEOUT	Sets resource time-out to *n* number of seconds. **Parameters:** *n* Default is 0 for Oracle (i.e., no time-out), 60 seconds for SQL Server, and database-dependent for other drivers. (For example, the default for MySQL is 50 seconds, and to change this, you must change the configuration for MySQL.) See "SSQL_TIMEOUT" on page 3-15.	All	Connection

Database-Specific Options (Continued)			
Option	**Description/parameters**	**Database driver(s)**	**Duration of setting**
SSQL_TRIMCHAR	Changes data type (*dty*) for character string conversions. **Parameters:** *dty* Default is 1, which is VARCHAR. See "SSQL_TRIMCHAR" on page 3-15.	VTX0	Connection
SSQL_TXN_ISOLEVEL	Sets the ODBC cursor isolation level. **Parameters:** SSQL_TXN_READ_COMMITTED SSQL_TXN_READ_UNCOMMITTED SSQL_TXN_REPEATABLE_READ SSQL_TXN_SERIALIZABLE See "SSQL_TXN_ISOLEVEL" on page 3-16.	VTX11 VTX12_ODBC VTX12_SQLNATIVE	Connection
SSQL_USEDB	Specifies a database name for connections strings in subsequent %SSC_OPEN calls. **Parameters:** *dbname* No default. See "SSQL_USEDB" on page 3-16.	VTX2 VTX12_ODBC VTX12_SQLNATIVE	Connection

SSQL_CACHE_CHAIN

On Windows and UNIX, this option instructs %SSC_RELEASE to cache database connections and preserves the cache when chaining to other programs. When SSQL_CACHE_CHAIN is in effect, any program you chain to must still call %SSC_INIT and %SSC_CONNECT for each database connection, but %SSC_CONNECT checks connections cached by %SSC_RELEASE and uses one if possible. For a cached connection to be used, however, the connection string passed to %SSC_CONNECT must be identical to the connection string used for the cached connection. Note the following:

- This option is identical to SSQL_CACHE_CONNECTION, except that with SSQL_CACHE_CONNECTION, cached connections are closed when a program chains.
- You can use the *force_release* argument for %SSC_RELEASE to override this setting. See %SSC_RELEASE on page 3-50 for more information.
- On OpenVMS, this option causes errors.

SSQL_CACHE_CONNECTION

This option is identical to SSQL_CACHE_CHAIN except that with this option, cached connections are closed when a program chains. Note that it is generally better to use SSQL_CACHE_CHAIN and maintain the connection cache. Use SSQL_CACHE_CONNECTION only if connection strings for the new program (the program that is assuming control) are different than the connection strings in the original program.

On OpenVMS, this option causes errors.

SSQL_CMD_SCROLL

Determines which row will be retrieved in the next fetch for a scrolling cursor (a cursor opened with one of the %SSC_OPEN options that begin with SSQL_SCROLL):

SSQL_SCROLL_CURRENT The row at the current cursor position

SSQL_SCROLL_FIRST The first row in the result set

SSQL_SCROLL_LAST The last row in the result set

SSQL_SCROLL_NEXT The row that follows the row at the current cursor position (default)

SSQL_SCROLL_PRIOR The row that proceeds the row at the current position

SSQL_SCROLL_ABSOLUTE±*n* A specific row in the result set where *n* is the number of the row you want retrieved with the next fetch. If *n* is a positive number, the *n*th row from the beginning of the result set will be fetched. If *n* is a negative number, the *n*th row from the end of the result set will be fetched. If *n* is 0, the row at the current cursor position will be the next row fetched. The following, for example, retrieves data from the third row from the end of the result set:

```
sts=(%ssc_cmd(dbchn, cur2, SSQL_CMD_SCROLL,
&             SSQL_SCROLL_ABSOLUTE-3))
```

SSQL_SCROLL_RELATIVE±*n* A specific row relative to the current cursor position where *n* is the number of the rows beyond the current cursor position if *n* is positive. If negative, the *n*th row before the current cursor position will be fetched. If *n* is 0, the row at the current cursor position will be the next row fetched. The following, for example, retrieves data from the next row (the row that follows the current cursor position):

```
sts=(%ssc_cmd(dbchn, cur2, SSQL_CMD_SCROLL,
&             SSQL_SCROLL_RELATIVE+1))
```

For more information on scrolling cursors, see "Cursor types" on page 2-29 and "Specifying a cursor type" on page 2-31.

SSQL_CURSOR_TYPE

This option sets the database cursor type for subsequent %SSC_OPEN calls:

SSQL_CURSOR_DEFAULT	Default cursor (same as SSQL_CURSOR_FORWARD_ONLY)
SSQL_CURSOR_DYNAMIC	Dynamic cursor
SSQL_CURSOR_FORWARD_ONLY	Forward-only cursor
SSQL_CURSOR_KEYSET_DRIVEN	Keyset-driven cursor
SSQL_CURSOR_STATIC	Static cursor

For information on cursors, including cursor types and how to set them, see "Cursors" on page 2-27 and your database documentation. Additionally, note the following:

- None of these options create a scrolling cursor unless you also use one of the scrolling options for %SSC_OPEN (options that begin with SSQL_SCROLL). For example, if you set SSQL_CURSOR_DYNAMIC in an %SSC_CMD call and don't use any of the scrolling options in the %SSC_OPEN call, the cursor will be dynamic (if supported by the database) and forward-only.
- The %SSC_OPEN scrolling options (except SSQL_SCROLL) override the SSQL_CURSOR_TYPE options. So if you set SSQL_CURSOR_STATIC and then set SSQL_SCROLL_DYNAMIC in the %SSC_OPEN call, the cursor will be opened as a scrolling dynamic cursor. (However, if you set SSQL_CURSOR_STATIC in an %SSC_CMD call and then set SSQL_SCROLL in and %SSC_OPEN call, the cursor will be opened as a scrolling static cursor.)

SSQL_KEEP_OPEN

On Windows and UNIX, this option ensures that the Synergy runtime does not shut down connections in a program chain. If you use this option, do not use %SSC_INIT or %SSC_CONNECT in a program you're chaining to.

On OpenVMS, this options causes errors.

SSQL_LANGVER

Specifies the version of the Oracle parser to be used.

OCI_NTV_SYNTAX	Instructs the database driver to use the default parser for the Oracle database the program is connected to.
OCI_V7_SYNTAX	Instructs the database driver to use Oracle7 syntax. This is the default.
OCI_V8_SYNTAX	Instructs the database driver to use Oracle8 syntax.

SSQL_NEW_BLOBS

Instructs Oracle to use BLOB or CLOB data rather than LONG RAW or LONG data.

SSQL_ODBC_AUTOCOMMIT

Enables you to turn autocommit mode on or off.

SSQL_OLD_ZONEDDATE

Restores pre–version 7 behavior when date fields are retrieved into decimal fields with %SSC_MOVE. When SSQL_OLD_ZONEDDATE is set, decimal fields are treated as alpha fields when used in conjunction with an %SSC_OPTION date-time mask. For this pre–version 7 behavior to take effect, this option must be set before using %SSC_OPEN.

SSQL_ONEPID

Instructs the database to use a single process for operations. We don't recommend using this option, though it does work with Sybase. And, while it may eliminate some deadlock errors, it can also adversely affect performance. Additionally, nested queries (for example, combined fetch and update operations) won't work if it is set.

SSQL_RAWDATE

Returns date/time untouched (doesn't convert date/time to the data type of the defined variable).

SSQL_RETURN_ROWID

Determines whether a row ID will be returned for each SQL statement when using VTX0 (Oracle), or VTX5 (Informix). By default a row ID will not be returned.

SSQL_RO_CURSOR

Instructs SQL Connection to use fast-forward cursors (SQL_CO_FFO) if possible, so we recommend using this option. However, this option applies only if your application uses VTX12_ODBC or VTX12_SQLNATIVE, uses static or dynamic cursors, and does not update the database.

SSQL_SQL_BULK_INSERT

If you use array variables in an %SSC_EXECUTE call for a SQL Server database, this option enables you to use bulk inserts, which improve performance. However, note that you should use this option only when no concurrent database activity is expected for the affected rows. You can use this option, for instance, to improve performance when loading initial data into database tables. Note the following:

- The bulk insert feature does not use the full SQL statement passed in %SSC_OPEN. It uses the table information and the bind variables (:1, :2, etc.), but ignores the rest of the statement.
- There must be as many bind variables as there are columns in the table, including any timestamp column. The first bind variable corresponds to the first column in the table, the second bind variable corresponds to the second column, and so forth.
- If there is a timestamp column in the table, you must pass an empty string ("") for this column if there is no data for the row.
- Dates inserted using bulk insert must have the "YYYY-MM-DD" format.
- SQL statements for bulk inserts cannot contain functions.

SSQL_SYB_BLANK

Specifies that a single blank VARCHAR data field should return a space instead of null on Sybase.

SSQL_TIMEOUT

Specifies the time-out for resource waits, such as locking on the database. SSQL_TIMEOUT sets the number of seconds to wait before returning a resource error (such as a locked row). Note that some databases (Oracle, for example) do not support a timeout setting. And note that on any resource error, the timeout value is ignored and the database returns an error immediately.

SSQL_TRIMCHAR

Defines the Oracle data type used for character conversions from SQL Connection to Oracle database char and varchar columns. *Dty* is the Oracle data type to use; see your Oracle documentation for the values of different Oracle data types.

1 Instructs SQL Connection to use the VARCHAR data type when converting to a char field so that no trailing blanks are stored. If a field consists entirely of spaces, the field will be stored as null. This is the default.

96 Instructs SQL Connection to use the CHAR data type when converting string data so that trailing blanks are stored.

SSQL_TXN_ISOLEVEL

Specifies the ODBC cursor isolation level.

SSQL_TXN_READ_COMMITTED

SSQL_TXN_READ_UNCOMMITTED

SSQL_TXN_REPEATABLE_READ

SSQL_TXN_SERIALIZABLE

The default is database dependent. See your database documentation for more information.

For SQL Server, if your application does not update the database, we recommend that you set the %SSC_CMD option SSQL_TXN_ISOLEVEL to SSQL_TXN_READ_UNCOMMITTED.

SSQL_USEDB

Specifies a default database name (*database_name*) for connect strings passed in subsequent %SSC_OPEN calls when connecting to SQL Server or Sybase.

If you've submitted an SQL statement and want to use it for another database, use this option (rather than a USE DATABASE command) to specify the new database. (A USE DATABASE command generally causes errors in this situation because it allows cached statements for the original database to be used rather than submitting the statement anew to the specified database.)

The following example from the **exam_fetch.dbl** example program uses %SSC_CMD to select the SQL Server database "PUBS".

```
if (%ssc_cmd(dbchn, cur3, SSQL_USEDB, "pubs")) exit
```

%SSC_COMMIT – Start or commit a transaction

WT	WN	U	V

value = %SSC_COMMIT(*dbchannel[, mode]*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

SSQL_FAILURE Failure

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

mode

(optional) Indicates whether to start and/or commit a transaction. (The default differs from one database to another, and *mode* is ignored if a database does not support different transaction states.) (**n**)

SSQL_TXON Start a transaction. If a transaction already exists for *dbchannel*, commit that transaction first.

SSQL_TXOFF Commit the transaction for *dbchannel*.

Discussion

Depending on *mode*, %SSC_COMMIT either commits a transaction, starts a new transaction, or both. If it commits a transaction, it hard-closes any associated cursors.

- If you pass SSQL_TXOFF, %SSC_COMMIT commits the transaction for *dbchannel*.
- If you pass SSQL_TXON and your database supports explicit transactions, %SSC_COMMIT starts a new transaction for *dbchannel*. If a transaction already exists for *dbchannel*, %SSC_COMMIT first commits that transaction and then starts the new one.

For databases, such as SQL Server, that support both explicit and implicit transactions, %SSC_COMMIT starts an explicit transaction. For databases that don't support explicit transactions (such as Oracle), %SSC_COMMIT does not start a transaction. In this case,

- the first data access (DML) operation (%SSC_OPEN, %SSC_EXECECUTE, %SSC_EXECIO) starts an implicit transaction.
- %SSC_COMMIT commits the data to the database, and SSQL_TXON has the same affect as SSQL_TXOFF.

For more information, see "Transactions and Autocommit" on page 2-49.

Examples

The following example demonstrates the use of %SSC_COMMIT.

```
sqlp = "INSERT INTO org1 (deptnum, deptname, manager,"
  &      " division, stdate, budget) VALUES (:1,:2,:3,:4,"
  &      ":5,:6)"
if (%ssc_commit(dbchn, SSQL_TXON))      ;Start the
  goto err_exit                         ; transaction
if (%ssc_open(dbchn, cur2, sqlp, SSQL_NONSEL,
  &             SSQL_STANDARD, 6, deptnum, deptname,
  &             manager, division, stdate, budget))
  goto err_exit

for ix from 1 thru MX_REC               ;Do insert
  begin
    call load_data                      ;Load data to bind
                                        ; area
                                        ;Execute insert
                                        ; statement
    if (%ssc_execute(dbchn, cur2, SSQL_STANDARD))
      goto err_exit
  end
if (%ssc_sclose(dbchn, cur2))
  goto err_exit
if (%ssc_commit(dbchn, SSQL_TXOFF))     ;Commit the
  goto err_exit                         ; change
```

%SSC_CONNECT – Connect to a database channel

WT	WN	U	V

value = %SSC_CONNECT(*dbchannel, constring*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

SSQL_FAILURE Failure

Arguments

dbchannel

The intended database channel. The range is 1 to 100 on Windows, and 1 to 7 on UNIX and OpenVMS. (**n**)

constring

A connection string. This argument varies according to database and configuration, but has a maximum size of 256. (**a**)

Discussion

%SSC_CONNECT connects to a database channel initialized by %SSC_INIT. (See "Database connections" on page 2-5 for more information.)

Note that every call to %SSC_CONNECT should be paired with a call to %SSC_RELEASE, unless you use the %SSC_CMD option SSQL_KEEP_OPEN. (For information on SSQL_KEEP_OPEN, see %SSC_CMD on page 3-7.)

On OpenVMS, if a channel is not explicitly released with %SSC_RELEASE, the associated license will not be released. Use the logging available with the SSQLLOG environment variable to verify that there is an %SSC_RELEASE call for every %SSC_CONNECT call. For information on SSQLLOG logging, see "SQL Connection logging" on page 5-5.

For information on connect strings you can pass as *constring*, see "Building Connect Strings" on page 2-16.

If the %SSC_CMD option SSQL_KEEP_OPEN is in force, passing the number for a database channel (*dbchannel*) that was opened by a prior program in a program chain will cause a runtime error.

Examples

The following example uses a case statement to determine what connect string to use.

```
case (MY_SSQL_SYSTEM) of                ;MY_SSQL_SYSTEM - user defined
  begincase
SQLSRVR:  user = "VTX12_odbc:sa/manager"     ;Do setup of connect string
ORACLE:  user = "VTX0_10:sa/manager"
  endcase
else
  user = "sa/manager"                  ;Default database connection
if (%ssc_connect(dbchn, user))
  goto err_exit
```

The next example tests for a SQL Server database (SSQL_DID_SQLSRV).

```
user = "sa/manager"                     ;Default database
if (%ssc_connect(dbchn, user))          ; connection
  goto err_exit
if (%ssc_getdbid(dbchn, dbid))          ;Get the database ID
  goto err_exit
if (dbid.eq.SSQL_DID_SQLSRV)
  if (%ssc_cmd(dbchn, SSQL_USEDB, "synergex"))
    goto err_exit
```

%SSC_DEFINE – Define host variables for the SELECT statement

WT	WN	U	V

value = %SSC_DEFINE(*dbchannel, dbcursor, numvars, var[, …]*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

SSQL_FAILURE Failure

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

dbcursor

The logical cursor number within the range from 1 through the maximum number specified by *maxcur* during %SSC_INIT. The cursor must have been opened by %SSC_OPEN. (**n**)

numvars

The number of host variables in the SELECT statement, up to the maximum number of columns specified by the *maxcol* argument in %SSC_INIT. If *numvars* is negative, the define variables are overwritten rather than appended. (**n**)

var

One to *numvars* host variables. The number of *var* variables passed must equal the value of *numvars*. (**a**, **n**, or String)

Discussion

%SSC_DEFINE defines host variables to variables specified in a SELECT statement. Depending on the *numvars* setting, the definitions are either appended to the variables for the SELECT statement, or the SELECT statement variables are overwritten with the host variable definitions. %SSC_DEFINE affects only those variables that are used when %SSC_EXECUTE is called.

The total number of defined variables used by %SSC_MOVE must match the number of columns defined by the SELECT statement for an open cursor. For example, if there are 20 host variables in the SELECT statement, you could use one %SSC_DEFINE to append definitions for all 20 variables, or two %SSC_DEFINEs with 10 variables each, or any other combination as long as the

number of variables defined by %SSC_DEFINE exactly matches the number of variables described by %SSC_OPEN in the SELECT statement before %SSC_MOVE is called:

- %SSC_DEFINE generates an error if the number of variables defined by one or more %SSC_DEFINEs exceeds the number of columns defined in the SELECT statement.
- %SSC_MOVE generates an error if the number of variables defined by all calls to %SSC_DEFINE does not exactly match the number required by the SQL statement used in %SSC_OPEN.

You can use %SSC_DESCSQL if you are uncertain about the number of variables to define (for example, if you use a SELECT * statement). For more information, see %SSC_DESCSQL on page 3-23. For more information on defining variables, see "Defining variables" on page 2-33.

Note the following:

- The String data type (System.String) is not supported for *var* for array-based operations, and is not supported for Synergy .NET.
- Do not use the Synergy DBL INIT statement for a String variable or a record that includes a String variable after passing that variable in a call to %SSC_DEFINE (unless you immediately pass it again in another call to %SSC_DEFINE). This will cause a runtime error.

If you use ^VARARGARRAY, note that *numvars* is the last declared argument for this routine.

Examples

The following example shows how to define four Synergy DBL variables associated with *cur1* and two variables associated with *cur2*.

```
record order_rec
       ord_num            ,d6
       ord_cust           ,d6
       ord_odate          ,a10
       ord_sdate          ,a10
record customer_rec
       cust_num           ,d6
       cust_name          ,a30
.
.
.
sqlp = "SELECT or_number, or_customer, or_date, orsdate FROM orders"
if (%ssc_open(dbchn, cur1, sqlp, SSQL_SELECT))
  goto err_exit
if (%ssc_define(dbchn, cur1, 4, ord_num, ord_cust, ord_odate, ord_sdate))
  goto err_exit

sqlp = "SELECT or_number, or_customer FROM orders WHERE or_customer = 101"
if (%ssc_open(dbchn, cur2, sqlp, SSQL_SELECT))
  goto err_exit
if (%ssc_define(dbchn, cur2, 2, cust_num, cust_name)
  goto err_exit
```

%SSC_DESCSQL – Describe an SQL statement

WT	WN	U	V

value = %SSC_DESCSQL(*dbchannel, dbcursor, numvars, description*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

SSQL_FAILURE Failure

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

dbcursor

The logical cursor number within the range from 1 through the maximum number specified by *maxcur* during %SSC_INIT. The cursor must have been opened by %SSC_OPEN. (**n**)

numvars

The maximum number of variables that can be returned in the *description* array. (**n**)

description

A returned record description. See the **ssc_desc** record in Examples below for the record layout. (**a**)

Discussion

This routine provides a way of finding out the number of variables you should define with an %SSC_DEFINE, and describes an SQL statement associated with the specified cursor.

With %SSC_DEFINE you must define the same number of variables as are defined by the number of SELECT columns, although you may not always know this number ahead of time. For example, when you use a SELECT * statement, %SSC_DESCSQL can determine the number of columns returned.

If the number of variables specified in the *description* argument is less than the number of columns defined in the SELECT statement, only the number of specified variables will be loaded.

The data fields returned are the returned values from the database and do not always make sense. For example, Oracle won't always return information in **var_len** for an integer that is 1, 2, 4, or 8 bytes.

Examples

The following example shows an SQL statement description layout in which **var_descs** is the layout of each column description.

```
.define MX_VARS      ,255
record ssc_desc
  var_nmbr           ,d3           ;Number of variables used
  group var_descs    ,[MX_VARS] a
    var_name         ,a30
    var_type         ,d2           ;Possible var_types are the following:
    var_len          ,d5           ;   0-integer
    var_dec          ,d2           ;   1-char
  end group                        ;   2-number
                                   ;   3-null-terminated char
                                   ;   4-packed decimal
                                   ;   5-zoned decimal
                                   ;   8-float
                                   ;   9-varchar
                                   ;   10-large binary object (blob)
                                   ;   11-large character object (clob)
                                   ;   12-datetime
                                   ;   80-varlen blob
                                   ;   81-Unicode char UTF-8 format
                                   ;   82-Unicode char UTF-16 format
                                   ;   83-Unicode char UCS-2 format
                                   ;   84-Unicode char UCS-4 format
                                   ;   85-Unicode LOB UTF-8 format
                                   ;   86-Unicode varchar UTF-8 format
                                   ;   87-Unicode varchar UTF-16 format
                                   ;   88-Unicode varchar UCS-2 format
                                   ;   89-Unicode varchar UCS-4 format
                                   ;   99-binary
                                   ; If var_type is an integer, var_len
                                   ; will always return the value of 10
                                   ;
                               ;Get SQL variable descriptions
sqlstm = "SELECT * FROM org WHERE deptnum = :1"
if (%ssc_open(dbchn, cur2, sqlstm, SSQL_SELECT,
  &           SSQL_STANDARD, 1, deptnum))
  goto err_exit
```

```
                              ;Open cursor #3 with
                              ; an SQL SELECT statement
if (%ssc_descsql(dbchn, cur2, MX_VARS, ssc_desc))
  goto err_exit
for ix from 1 thru var_nmbr  ;Display them
  begin
    sqlvar = var_descs(ix)
    display(1, "COL #", %string(ix), ": ",
  &            var_name, %string(var_type), " ",
  &            %string(var_len))
    writes(1, "COL #", %string(ix), ": ", var_name, %string(var_type),
  &        " ", %string(var_len), ".", %string(var_dec))
  end
;Define for maximum variables and control actual number of variables
; with var_nmbr
sts=%ssc_define(dbchn,cur2,var_nmbr,var1,var2,var3,var4,var5,var6,var7,
  &             var8,var9,var10,...varn)
```

%SSC_EXECIO – Execute a stored procedure with I/O parameters

WT	WN	U	V

value = %SSC_EXECIO(*dbchannel, dbcursor, [ncount], [numvars][, type, var, arg][, …]*)

Return value

value

This function returns an integer result, which is either the return result of a stored procedure, an error code returned by the database, or one of the following. (Negative values are considered an error.) (**i**)

SSQL_NORMAL Success

SSQL_FAILURE Failure

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

dbcursor

The logical cursor number within the range from 1 through the maximum number specified by *maxcur* in the %SSC_INIT call. The cursor must have been opened by %SSC_OPEN. (**n**)

ncount

(optional) The number of rows to execute. The default value is 1. Only certain databases (Oracle, SQL Server, and Sybase) support multirow operations. If this argument is used with a database that does not support multirow move, *ncount* must be set to 1. (**n**)

numvars

(optional) The number of host variables for sending to or receiving from the stored procedure. (The *maxcol* argument, as specified in %SSC_INIT, specifies the maximum value for *numvars*.) If you pass *numvars*, you must also include the *type*, *var*, and *arg* arguments. The number of times you include the *type*, *var*, *arg* series must equal the value of *numvars*. (**n**)

type

(optional) The type of the first host variable. (This is required if *numvars* is passed.) Note that if you pass SSQL_EXBINARY, you must also pass one of the other types and connect the two with a plus sign (+)—for example, SSQL_EXBINARY+SSQL_OUTPUT. (**n**)

SSQL_INPUT	Input
SSQL_OUTPUT	Output
SSQL_INOUT	Input and output
SSQL_OUTDATE	Output date field (to conform to pre-7.1 zoned conversion rules)
SSQL_EXBINARY	A binary column (instructs %SSC_EXECIO to leave the data as is; see the Discussion below)

var

(optional) The first host variable. (This is required if *numvars* is passed.) (**a**, **n**, or String)

arg

(optional) The name of the argument in the stored procedure. (This is required if *numvars* is passed.) (**a**)

Discussion

%SSC_EXECIO executes stored procedures that use input/output parameters. It does not set cursor properties and does not accept a result set. For more information on using stored procedures with SQL Connection, see "Stored Procedures" on page 2-51.

Note the following:

- When retrieving data into alpha fields, %SSC_EXECIO converts binary zeros to spaces and trims trailing spaces unless you use the SSQL_EXBINARY option. If you use this option, %SSC_EXECIO passes the data as is.
- When sending alpha variable data to a SQL Server or Oracle database, %SSC_EXECIO trims trailing spaces unless you use the SSQL_EXBINARY option. If you use this option, %SSC_EXECIO preserves the original length.
- Using %SSC_INDICATOR for SSQL_INOUT parameters is not possible because the indicator used internally in SQL Connection cannot be set to –1, or the INOUT bound data would be interpreted as null. SQL Connection depends on the initial state of the indicator variable set to –1 to be able to detect rows that are null or not fetched.

- Be careful not to confuse the return value with the SSQL_NOMORE return value for %SSC_MOVE. If you need to know if the rows were actually transferred as a result of a stored procedure reading data, you should return the row count (or number of rows returned) as one of your passed parameters. This way you can tell if your stored procedure was successful. %SSC_EXECIO differs from %SSC_EXECUTE and %SSC_MOVE in this area regarding row count.
- If you pass *ncount* as a number greater than 1, you must use arrays for the host/bind variables; otherwise, an error is generated.
- Do not use a String (System.String) variable for *var* for SSQL_OUTPUT or SSQL_INOUT if the returned data could be larger than the size of the variable.

.NET

- The String data type (System.String) is not supported in Synergy .NET.

- If you use ^VARARGARRAY, note that *numvars* is the last declared argument for this routine.

Examples

The Connectivity Series distribution includes example programs (in the connect\synsqlx directory) that use %SSC_EXECIO with stored procedures:

- For an Oracle example, see **stp_ora.dbl**.
- For SQL Server examples, see **stp_odbc.dbl**.
- For MySQL, see **stp_mysql.dbl**.

%SSC_EXECUTE – Execute a non-SELECT statement (no I/O parameters)

WT	WN	U	V

value = %SSC_EXECUTE(*dbchannel, dbcursor, [option], [ncount][, row_count]*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

SSQL_FAILURE Failure

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

dbcursor

The logical cursor number within the range from 1 through the maximum number specified by *maxcur* during %SSC_INIT. The cursor must have been opened by %SSC_OPEN. (**n**)

option

(optional) The type of operation. If you specify SSQL_LARGECOL, you can combine it with one of the other options by joining the options with a plus sign (+)—for example, SSQL_POSITION+SSQL_LARGECOL. (**n**)

SSQL_LARGECOL Use this if you'll use %SSC_LARGECOL for a large binary or character column. (See the Discussion below.)

SSQL_POSITION Use this for a linked cursor (i.e., if you use %SSC_SQLLINK).

SSQL_STANDARD Use this for a non-linked cursor (i.e., if you don't use %SSC_SQLLINK). (default)

ncount

(optional) The number of rows to execute. The default value is 1. (**n**)

Only certain databases (Oracle, Sybase, and SQL Server) support multirow operations. If this argument is used with databases that do not support multirow moves, *ncount* must be set to 1 (the default value).

row_count

(optional) Returned number of rows affected by executing the SQL statement associated with *dbcursor*. This count is valid only when *value* is returned as SSQL_NORMAL. If *value* is SSQL_NORMAL, a return value of zero for *row_count* indicates that there are no matches for the statement's WHERE clause. (**n**)

Discussion

%SSC_EXECUTE executes a non-SELECT statement and returns the result. (It does not set cursor properties and does not accept a result set.) This function is typically used to insert, delete, and update data. It can also be used to run a non-parameterized stored procedures in some cases (see "Invoking stored procedures" on page 2-52).

Note the following:

- Data is physically bound at %SSC_EXECUTE time (unlike SELECT statements where data is actually bound at %SSC_OPEN or %SSC_REBIND time).
- Input variables can be bound using %SSC_OPEN or %SSC_STRDEF. In database terms, this could be considered a direct execute without a previous prepare operation.
- If %SSC_EXECUTE follows an %SSC_SQLLINK call (i.e., if you pass SSQL_POSITION), it executes the linked statement rather than the original SELECT statement.
- When putting data for a large binary or character column, use the SSQL_LARGECOL option in conjunction with %SSC_LARGECOL. This instructs %SSC_EXECUTE to use the data in the string argument (*buf*) passed in the call to %SSC_LARGECOL. (If you don't use SSQL_LARGECOL and %SSC_LARGECOL, you can put no more than 65,533 bytes for a column.) See %SSC_LARGECOL on page 3-38.

If you have submitted a query and want to use the query for another database, do not use %SSC_EXECUTE to specify a different database. Instead use SSQL_USEDB, an %SSC_CMD option. (See SSQL_USEDB on page 3-16.) %SSC_EXECUTE generally causes errors in this situation because it allows cached statements from the original database to be used rather than submitting the statement to the newly specified database.

Examples

The following examples execute non-SELECT SQL statements. Note that for the first example, if the SQL statement is valid, but no rows meet the WHERE clause criteria, the function will return SSQL_NORMAL, and rows_returned will be returned as zero.

```
sqlp = "UPDATE customers SET cust_limit = 5000 WHERE cust_rtype > 1"

if (%ssc_open(dbchn, cur1, sqlp, SSQL_NONSEL))
  goto err_exit
```

```
if (%ssc_execute(dbchn, cur1, SSQL_STANDARD,, rows_returned))
  goto err_exit

if (rows_returned) ;If any row met the "cust_rtype" criterion...
.
.
.
```

The next example drops a table named org:

```
if (%ssc_open(dbchn, cur1, "DROP TABLE org", SSQL_NONSEL))
  goto err_exit
if (%ssc_execute(dbchn, cur1, SSQL_STANDARD))
  goto err_exit
```

The following example is for SQL Server:

```
sqlp = "CREATE TABLE org1 (deptnum int NOT NULL, deptname"
  &      " char(6) NOT NULL, manager int NOT NULL, division"
  &      " char(15) NOT NULL, stdate datetime, budget numeric)"
if (%ssc_open(dbchn, cur1, sqlp, SSQL_NONSEL))
  goto err_exit
                                        ;Execute the SQL in
                                        ; standard mode
if (%ssc_execute(dbchn, cur1, SSQL_STANDARD))
  goto err_exit
```

The following example is for Oracle:

```
sts = %ssc_commit(dbchn, SSQL_TXON)       ;Begin transaction mode
sqlp = "INSERT INTO org1 (deptnum, deptname, manager, division, "
  &    "hrdate, salary) VALUES (:1,:2,:3,:4,to_date(:5,"MM/DD/YYYY"),:6)"
                                          ;Open another cursor
if (%ssc_open(dbchn, cur2, sqlp, SSQL_NONSEL, SSQL_STANDARD, 6,
  &             deptnum, deptname, manager,
  &             division, hrdate, salary))
  goto err_exit

for ix from 1 thru MX_REC                 ;Do insert
begin                                     ;Load data to bind area
    deptnum = s_deptnum(ix)
    deptname = s_deptname(ix)
    manager = s_manager(ix)
    division = s_division(ix)
    hrdate = s_hrdate(ix)
    salary = s_salary(ix)
                                          ;Execute insert statement
```

```
    if (%ssc_execute(dbchn, cur2, SSQL_STANDARD))
      goto err_exit
  end
sts = %ssc_commit(dbchn, SSQL_TXOFF)   ;Commit the change and end
                                       ; transaction mode
if (%ssc_close(dbchn, cur2))
  goto err_exit
```

For another example of %SSC_EXECUTE, one that includes an example of a bulk insert, see **exam_create_table.dbl**, which is in the connect\synsqlx subdirectory of the main Synergy/DE installation directory.

%SSC_INDICATOR – Retrieve indicator variables

WT	WN	U	V

value = %SSC_INDICATOR(*dbchannel, dbcursor, int_array, [ncount][, vars]*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

dbcursor

The logical cursor number within the range from 1 through the maximum number specified by *maxcur* during %SSC_INIT. The cursor must have been opened by %SSC_OPEN. (**n**)

int_array

A real array whose elements will be filled with SQL indicator variables. (**n**)

ncount

(optional) The row of a multirow operation whose status you want returned. The default value is 1. (**n**)

vars

(optional) The number of *vars* moved, up to a maximum of *int_array* elements. (**a** or **n**)

Discussion

The value for each element in *int_array* relays the status of either the defined variables in the output order specified by %SSC_DEFINE or %SSC_STRDEF, or the defined variables in the output and in/out order specified by %SSC_EXECIO.

Possible returned values for each element in *int_array*:

– 1	The column is null (or no data was moved in the last statement).
0	The column was moved successfully.
> 0	The data was truncated, and the non-truncated size is reported.

Examples

The following is an example of retrieving the status of two SELECT columns, **deptnum** and **deptname**. In particular, this example reports the status of moving *deptnum* into the host variable and whether *deptname* was returned as a null or the data was truncated.

```
sqlp = "SELECT deptnum, deptname FROM org WHERE deptnum = 1"
if (ssc_open(dbchn, cur2, sqlp, SSQL_SELECT, SSQL_STANDARD))
  goto error_exit
if (ssc_define(dbchn, cur2, 2, deptnum, deptname))
  goto error_exit
if (ssc_move(dbchn, cur2, 1))
  goto error_exit
if (ssc_indicator(dbchn, cur2, intarray))
  goto error_exit
writes(15, "status of deptnum is"+%string(intarray[1]))
writes(15, "status of deptname is"+%string(intarray[2]))
```

%SSC_INIT – Initialize a database channel

WT	WN	U	V

value = %SSC_INIT(*dbchannel, [maxcur], [maxcol], [bufsize][, dbcursor]*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

SSQL_FAILURE Failure

Arguments

dbchannel

An internal database channel to be used for a connection. The range is 1 to 100 on Windows, and 1 to 7 on UNIX and OpenVMS. (**n**)

maxcur

(optional) The number of logical cursors to allocate for the channel. The default is 128. (**n**)

maxcol

(optional) The maximum number of columns that can be returned from a query. The default is 254. (**n**)

bufsize

(optional) The internal buffer size (in bytes) used for prefetch caching. The default is 32768. (**n**)

dbcursor

(optional) The number of database cursors to allocate for the channel. The default is *maxcur*. (**n**)

Discussion

%SSC_INIT must be the first function call when using SQL Connection, except

- on OpenVMS where %INIT_SSQL must be called prior to %SSC_INIT.
- in a program you chain to when using the SSQL_KEEP_OPEN option for %SSC_CMD. In this case, do not use %SSC_INIT in the program you chain to.

%SSC_INIT initializes an SQL Connection session, establishes an internal structure containing information used by %SSC_CONNECT and other routines, and provides a method for defining communication with the low-level structures. The internal structure is accessed by *dbchannel*, which is analogous to a channel established in a Synergy DBL OPEN statement. See "Database connections" on page 2-5 for more information, and note the following:

- You can have up to seven concurrently open channels on UNIX and OpenVMS and up to 100 concurrently open channels on Windows (to accommodate multi-threading).
- We recommend using .DEFINE identifiers for channel numbers.
- SQL Connection assigns a logical cursor to each SQL Statement. If *maxcur* is set to a value that is greater than the *dbcursor* setting, SQL Connection is able to cache cursors by mapping multiple logical cursors to a single database cursor. Because logical cursors require less memory than database cursors, this improves performance. For optimal performance, set *maxcur* to the maximum number of SQL statements that will be open (and soft-closed) concurrently.
- If you use %SSC_SCLOSE, the value of *dbcursor* should be the maximum concurrent number of open cursors plus a percentage of the soft-closed cursors.
- Ideally, *maxcol* should be set to the largest number of columns of any table within the connected database.
- The maximum values for *maxcur*, *maxcol*, *bufsize*, and *dbcursor* are database dependent.
- We recommend that you test with *bufsize* set to a higher value than the default—for example, 65536. This may improve performance.
- If 0 is specified for *bufsize*, the size of at least one row will be used for the prefetch buffer.
- To conserve memory and resources, you can use values that are less than the defaults for *maxcur*, *maxcol*, *bufsize*, and *dbcursor*. Note that reducing the memory used by an application may enable it to support more concurrent users (see "Reducing memory and enabling more concurrent users" on page 2-56).
- If you specify a previously initialized channel in a call to %SSC_INIT and there is an open connection, the open connection will be closed unless SSQL_KEEP_OPEN is set. When SSQL_KEEP_OPEN is set, %SSC_INIT calls on previously opened channels are ignored.

Examples

The following is an example of SQL channel initialization using database channel 1 and setting 32 concurrent cursors, 100 maximum columns, and a 5000-byte prefetch buffer size.

```
dbchn = 1                                ;Use database channel 1
if (%ssc_init(dbchn, 32, 100, 5000))
  goto err_exit                 ;Initializes connection to use 32
                                ; concurrent cursors, 100 maximum columns,
                                ; and 5000-byte prefetch buffer size
if (%ssc_connect(dbchn, user))
  goto err_exit
```

%SSC_LARGECOL – Get or put a large binary or char column

WT	WN	U	V

value = %SSC_LARGECOL(*dbchannel, dbcursor, buf, opts, col, len[, binary]*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

SSQL_FAILURE Failure

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

dbcursor

The logical cursor number (within the range from 1 through the maximum number specified by *maxcur* during %SSC_INIT). The cursor must have been opened by %SSC_OPEN. (**n**)

buf

The data for the column. For a get, a new string is created for *buf*. For a put, the data to be put must be passed in *buf*. (String)

opts

The operation option. (**n**)

SSQL_LARGEGET Get large binary or character data.

SSQL_LARGEPUT Put large binary or character data.

col

The column position (zero based). (**n**)

len

For a get, this must be the length of the string returned into the host variable by %SSC_MOVE. For a put, this is the returned length of the put string. (**n**)

binary

(optional) If passed, this indicates that the column is a binary column, which prevents %SSC_LARGECOL from removing trailing characters. (**n**)

Discussion

%SSC_LARGECOL puts or gets large binary column (BLOB) or large character column (CLOB) data. In SQL Server, for example, these are VARCHAR(MAX) and VARBINARY(MAX) fields.

This routine takes a Synergy/DE String object and, for a get, creates a new String object. By using the Synergy/DE System.String data type, the object can exceed 65,535 bytes. Remember, though, that to access a String object that's greater than 65,535 bytes on a 32-bit system, you must range into the string. (See "String" in the "Defining Data" chapter of the *Synergy DBL Language Reference Manual*.)

To *get* large binary column or large character column data, do the following:

1. Specify the SSQL_LARGECOL and SSQL_SELECT options in the %SSC_OPEN call for the statement.
2. Use %SSC_DEFINE to set up an **i4** host variable for the column.
3. Call %SSC_MOVE. %SSC_MOVE retrieves the size of the column and saves it in the **i4** host variable.
4. After each %SSC_MOVE call, pass the **i4** variable as the *len* argument in the %SSC_LARGECOL call.

To *put* large binary column or large character column data, do the following:

1. Specify SSQL_NONSEL in the %SSC_OPEN call for the statement. (Don't specify SSQL_LARGECOL here. You'll do that in step 3.)
2. Before calling %SSC_EXECUTE, call %SSC_LARGECOL to set up the buffer (*buf*) that will be bound with %SSC_EXECUTE.
3. Call %SSC_EXECUTE, specifying SSQL_LARGECOL.

Examples

The following code segment demonstrates how to get VARCHAR(MAX) data from a SQL Server database.

```
record data
       buffer  ,string
       length  ,i4
.
.
.
sqlp = "select bcol from btab where btab_id = :1"
if (sts = %ssc_open(dbchn, cur1, sqlp, SSQL_SELECT,
  &                 SSQL_STANDARD + SSQL_LARGECOL, 1, id))
  goto err_exit
```

```
if (sts = %ssc_define(dbchn, cur1, 1, length)
  if (sts = %ssc_move(dbchn, cur1) != ssql_success)
    goto err_exit
  if (sts = %ssc_largecol(dbchn, cur1, buffer, SSQL_LARGEGET, 0, length))
      ;Where "0" (the col parameter) is the first column of bcol
    goto err_exit

  if (sts = %ssc_sclose(dbchn, cur1))
    goto err_exit
```

%SSC_MOVE – Fetch rows of data

WT	WN	U	V

value = %SSC_MOVE(*dbchannel, dbcursor, [ncount], [row_count], [compute_flg][, warn]*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

SSQL_FAILURE Failure

SSQL_NOMORE No more data found for current result set

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

dbcursor

The logical cursor number within the range from 1 through the maximum number specified by *maxcur* during %SSC_INIT. The cursor must have been opened by %SSC_OPEN. (**n**)

ncount

(optional) The number of rows to fetch. This argument defaults to 1 for databases that do not support multirow fetch. (**n**)

row_count

(optional) Returned number of rows actually fetched by the SQL statement associated with *dbcursor*. This count is valid only when *value* is returned as SSQL_NORMAL. (**n**)

compute_flg

This argument is no longer supported.

warn

(optional) A variable that is set to 1 if one or more rows return a warning status (such as “data columns truncated”). (**n**)

Discussion

%SSC_MOVE fetches one or more rows of data into host variables defined by %SSC_DEFINE or %SSC_STRDEF. For multirow fetches, %SSC_MOVE returns SSQL_NOMORE if not all requested rows are fetched. (If you request a four-row fetch, for example, but %SSC_MOVE is able to fetch only three rows, %SSC_MOVE returns SSQL_NOMORE.) You can use *row_count* to find out how many rows were actually fetched.

Note the following:

- %SSC_MOVE works only with SELECT statements and SQL Server stored procedures, so the %SSC_OPEN call that precedes %SSC_MOVE must set SSQL_SELECT. (%SSC_MOVE fetches rows for a SELECT cursor, even if there's an intervening call to %SSC_SQLLINK.)
- For large binary columns and large character columns, if you pass SSQL_LARGECOL in the %SSC_OPEN call, %SSC_MOVE returns the field length (rather than the data) into the host variables defined for the columns. You then use %SSC_LARGECOL calls to fetch the data. (If you don't pass SSQL_LARGECOL in the %SSC_OPEN call, %SSC_MOVE fetches the column as a 65,533-byte binary or char column. If the data is longer than 65,533 bytes, the data will be truncated.) See %SSC_LARGECOL on page 3-38 for more information.
- %SSC_MOVE can be used to fetch data from a SQL Server stored procedure result set. See "Invoking stored procedures" on page 2-52, and see **stp_sqlsrv.dbl** and **stp_sqlsrv2.dbl** for examples.

Examples

The following example shows how to move column data to a Synergy DBL data area.

```
sqlp = "SELECT deptnum, deptname"
  &     " FROM org WHERE deptnum = :1"
sts=%ssc_open(dbchn, cur2, sqlp, SSQL_SELECT, SSQL_STANDARD, 1, deptnum)
sts=%ssc_define(dbchn, cur2, 2, deptnum, deptname)
;  Get dnum to SELECT rows
display(g_terminal, "Enter Department Number: ")
reads(g_terminal, %a(dnum))
;  Do fetch and display rows to screen one row at a time
do forever
  begin
    sts = %ssc_move(dbchn, cur2, 1)
    if (sts.eq.SSQL_FAILURE) then             ;ERROR
      goto err_exit
    else if (sts.eq.SSQL_NOMORE)              ;EOF
      exitloop
    writes(g_terminal, %string(deptnum) + ", " + deptname)
  end
```

For an example of a single row fetch, see **exam_fetch.dbl**. For an example of a multirow fetch, see **exam_multirow_fetch.dbl**. These example files are in the connect\synsqlx subdirectory of the main Synergy/DE installation directory.

%SSC_OPEN – Open a cursor

WT	WN	U	V

value = %SSC_OPEN(*dbchannel, dbcursor, statement, type, [options], [numvars][, var, …]*)

Return value

value

This function returns one of the following. (**i**)

SSQL_NORMAL Success

SSQL_FAILURE Failure

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

dbcursor

The ID number for the cursor. This argument serves two purposes: it returns the ID number for the cursor for *statement*, and it determines whether %SSC_OPEN will simply open a new cursor for the statement or whether it will attempt to reuse a cursor. (**n**)

- If *dbcursor* is passed as a value of zero, %SSC_OPEN opens a new cursor and returns the number for that cursor in *dbcursor*.
- If *dbcursor* is passed as a non-zero value, %SSC_OPEN reuses the cursor if it can (see "Multiple cursors, dbcursor, and cursor reuse" on page 3-45) and returns the same cursor number. If %SSC_OPEN is unable to reuse the cursor, it hard closes it and opens a new cursor.

statement

An SQL statement. (**a**)

type

The SQL statement type: (**n**)

SSQL_SELECT SELECT statement or the name of a SELECT stored procedure.

SSQL_NONSEL Non-SELECT statement (such as INSERT, UPDATE, or DELETE).

options

(optional) Sets options used to configure a cursor. (All but SSQL_LARGECOL are useful only with cursors for SELECT statements.) You can pass more than one option by joining the options with a plus sign (+). For more information on the following options (including a table that lists which options can be used together in the same call), see "The options argument" on page 3-46. (**n**)

SSQL_SCROLL	Creates a scrolling cursor of the type specified with an SSQL_CURSOR option in a previous call to %SSC_CMD. If no SSQL_CURSOR option has been set, creates a cursor of the default type for the database. This can be used only with VTX0, VTX11, VTX12_ODBC, or VTX12_SQLNATIVE.
SSQL_SCROLL_DYNAMIC	Creates a scrolling dynamic cursor. This can be used only with VTX11, VTX12_ODBC, or VTX12_SQLNATIVE.
SSQL_SCROLL_KEYSET	Creates a scrolling keyset-driven cursor. This can be used only with VTX11, VTX12_ODBC, or VTX12_SQLNATIVE.
SSQL_SCROLL_READ_ONLY	Creates a scrolling static cursor. This can be used only with VTX11, VTX12_ODBC, or VTX12_SQLNATIVE.
SSQL_LARGECOL	Enables SQL Connection to use %SSC_LARGECOL to get or put large binary columns or large character columns. See %SSC_LARGECOL on page 3-38.
SSQL_FORUPDATE	Informs SQL Connection that the SQL statement (*statement*) passed to %SSC_OPEN contains a FOR UPDATE OF clause. This is required if *statement* contains a FOR UPDATE OF clause.
SSQL_ONECOL	Explicitly disables prefetch caching.
SSQL_POSITION	Creates a positioned cursor that is positioned at the first record that meets the criteria for the query.
SSQL_STANDARD	Creates a standard (non-positioned) cursor, and when used without other options for the *options* argument, enables prefetch caching.

numvars

(optional) The number of variables (*var*, …) bound to *statement*. This must be set to the number of *var* arguments passed. (**n**)

var

(optional) Host variable to be bound to *statement*. You can pass more than one *var* argument (by separating them with commas). The number of *var* arguments you pass must equal the number passed as *numvars*. For information on binding host variables, see "Data Mapping" on page 2-33. (**a**, **n**, or String)

Discussion

%SSC_OPEN opens a cursor and associates it with the passed SQL statement (*statement*). The cursor is opened on the database channel specified by *dbchannel*. This section discusses %SSC_OPEN options and issues, but for more information on cursors, including information on cursor types and specifying cursor behavior with %SSC_OPEN and %SSC_CMD cursor options, see "Cursors" on page 2-27. Note the following:

- The String data type (System.String) is not supported for *var* for array-based operations.
- Do not use %SSC_OPEN with SSQL_SELECT and a server-side cursor for non-cursor related stored procedures, and do not use these if the SQL statement includes the EXEC SQL command for a stored procedure that returns no rows (this may cause "No cursor" warnings). Instead, use %SSC_OPEN with SSQL_NONSEL (and %SSC_EXECUTE).
- If you use ^VARARGARRAY, note that *numvars* is the last declared argument for this routine.

Multiple cursors, *dbcursor*, and cursor reuse

You can open multiple cursors concurrently. The maximum number cursors you can open is set by the *maxcur* argument for %SSC_INIT (though the number of actual database cursors that can be open concurrently is set by the *dbcursor* argument for %SSC_INIT and is limited by your database's capacity).

Note that you may not need to open a new cursor for each SQL statement. If you're going to reuse the same operation soon, it's best to reuse a cursor. (When a cursor is reused, the application skips the initial step of processing the SQL statement, which is typically a very resource-intensive process.) See "Reusing cursors" on page 2-28.

- If you pass a value of 0 as *dbcursor* in an %SSC_OPEN call, SQL Connection automatically searches to see if there's already a cached cursor on the database that can be reused for the statement. If there is one, SQL Connection uses this cursor.
- If you pass a non-zero value as *dbcursor*, SQL Connection attempts to reuse the cursor that corresponds to the non-zero value.

Additionally, if you are fetching a row and you plan to perform a positioned update, you can use %SSC_SQLLINK to link the update statement to the open SELECT cursor rather than opening another cursor for the update statement. See %SSC_SQLLINK on page 3-56.

The *options* argument

The *options* argument enables you to specify

- a scrolling cursor type.
- whether the cursor will be positioned.
- whether the SQL statement passed to %SSC_OPEN contains a FOR UPDATE OF clause.
- how SQL Connection will retrieve or write large amounts of binary or char data. (See %SSC_LARGECOL on page 3-38.)
- whether SQL Connection will use prefetch caching.

The SSQL_SCROLL options for %SSC_OPEN specify scrolling cursor types. (For SQL Server, these are ODBC API cursor types.) With a scrolling cursor you can determine which row will be retrieved with the next fetch. (You do this by setting an SSQL_CMD_SCROLL option with %SSC_CMD; see SSQL_CMD_SCROLL on page 3-12.) Note the following:

- You can specify only one SSQL_SCROLL option in a call to %SSC_OPEN. (For information on which arguments can be passed together in an %SSC_OPEN call, see the table below.)
- The SSQL_SCROLL options for %SSC_OPEN override %SSC_CMD cursor type options if they conflict.
- The SSQL_SCROLL options for %SSC_OPEN, SSQL_LARGECOL, and SSQL_ONECOL disable prefetch caching. If you pass none of these, but you do pass SSQL_SELECT and SSQL_STANDARD, SQL Connection uses prefetch caching which increases fetch (%SSC_MOVE) performance. (See "Improving network performance with prefetch caching" on page 2-55 for more information.)
- %SSC_CMD cursor type options do not by themselves create scrolling cursors: you must also specify one of the SSQL_SCROLL options to get a scrolling cursor.

For descriptions of and information on specifying cursor types (using %SSC_OPEN and/or %SSC_CMD), including information on creating a forward-only cursor (there's no %SSC_OPEN option for this), see "Specifying a cursor type" on page 2-31.

For the other options (SSQL_LARGECOL, SSQL_FORUPDATE, etc.), note the following:

- If you use SELECT to select a row and you need to lock the row for update, use positioned mode by setting SSQL_POSITION+SSQL_FORUPDATE (unless you're accessing a SQL Server database—see the note in "Row locking" on page 2-42). Positioned cursors allow other positioned operations (such as statements with UPDATE WHERE CURRENT OF on some databases), but note that some databases, such as Oracle Rdb, do not allow positioned mode.
- If you use SSQL_FORUPDATE, your SQL statement must include a FOR UPDATE OF clause. If your database does not use FOR UPDATE OF to invoke row locking, SQL Connection converts the SQL statement to a statement that does invoke row locking. For example, if you are using a Sybase database, SQL Connection converts the SELECT FOR UPDATE statement to a SELECT FOR BROWSE statement. For more information, see "Updates and Locking" on page 2-42.

See the table below for information on which options can be combined in a single %SSC_OPEN call. For example:

```
sts=%ssc_open(dbchn, cur1, sqlp, SSQL_SELECT,
 &             SSQL_STANDARD+SSQL_LARGECOL)
```

	SSQL_STANDARD	SSQL_POSITION	SSQL_FORUPDATE	SSQL_LARGECOL	SSQL_ONECOL	SSQL_SCROLL	SSQL_SCROLL_READ_ONLY	SSQL_SCROLL_DYNAMIC	SSQL_SCROLL_KEYSET
SSQL_STANDARD		✓		✓	✓				
SSQL_POSITION	✓		✓	✓	✓			✓	✓
SSQL_FORUPDATE		✓		✓	✓			✓	✓
SSQL_LARGECOL	✓	✓	✓		✓	✓	✓	✓	✓
SSQL_ONECOL	✓	✓	✓	✓		✓	✓	✓	✓
SSQL_SCROLL				✓	✓				
SSQL_SCROLL_READ_ONLY				✓	✓				
SSQL_SCROLL_DYNAMIC		✓	✓	✓	✓				
SSQL_SCROLL_KEYSET		✓	✓	✓	✓				

Examples

The following example opens three SQL statement cursors simultaneously.

```
if (%ssc_connect(dbchn, user))    ;Connects to database
  goto err_exit
       ;Open cursor #1
sqlp = "SELECT deptnum, deptname FROM org WHERE deptnum"
  &      " = 10"
if (%ssc_open(dbchn, cur1, sqlp, SSQL_SELECT, SSQL_STANDARD))
  goto err_exit
       ;Open cursor #2 where bind1 matches with :1
sqlstm = "SELECT deptnum, deptname, manager, division"
  &      " FROM org WHERE deptnum = :1"
```

```
if (%ssc_open(dbchn, cur2, sqlstm, SSQL_SELECT,
  &   SSQL_STANDARD, 1, bind1))
  goto err_exit
       ;Open cursor #3
sqlp = "INSERT INTO org (deptnum, deptname, manager,"
  &    " division, stdate, budget) VALUES (:1,:2,:3,:4,"
  &    " :5,:6)"
if (%ssc_open(dbchn, cur3, sqlp, SSQL_NONSEL,
  &   SSQL_STANDARD, 6, deptnum, deptname, manager,
  &   division, stdate, budget))
  goto err_exit
;where deptnum matches with :1, deptname matches with :2, etc.
```

For another example, see **exam_fetch.dbl**, which is in the connect\synsqlx subdirectory of the main Synergy/DE installation directory.

%SSC_REBIND – Rebind host variables for a new query

WT	WN	U	V

value = %SSC_REBIND(*dbchannel, dbcursor*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

SSQL_FAILURE Failure

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

dbcursor

The logical cursor number within the range from 1 through the maximum number specified by *maxcur* during %SSC_INIT. The cursor must have been opened by %SSC_OPEN. (**n**)

Discussion

%SSC_REBIND resubmits the SELECT statement for a SELECT cursor (SSQL_SELECT) and updates bind variables for the statement. Note that %SSC_REBIND

- does not update defined variables.
- does not resubmit statements linked to a SELECT cursor (by using %SSC_SQLLINK). If there is an intervening %SSC_SQLLINK call between the %SSC_OPEN that creates the SELECT cursor and the %SSC_REBIND, the %SSC_REBIND call will resubmit the SELECT statement defined in the %SSC_OPEN call.

Because %SSC_REBIND does not reprocess defined variables, it is more efficient when resubmitting a SELECT statement than using another %SSC_OPEN call, which would reparse the entire statement.

Examples

For an example of rebinding host variables, see **exam_fetch_update.dbl**, which is in the connect\synsqlx subdirectory of the main Synergy/DE installation directory.

%SSC_RELEASE – Release a database channel

WT	WN	U	V

value = %SSC_RELEASE(*dbchannel[, force_release]*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

SSQL_FAILURE Failure

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

force_release

(optional) Overrides the %SSC_CMD options SSQL_CACHE_CHAIN and SSQL_CACHE_CONNECTION when passed as non-zero. (**n**)

Discussion

%SSC_RELEASE closes all logical cursors, rolls back pending transactions, and frees the database channel number. It also closes the associated database connection unless one of the following %SSC_CMD options is set. Note that these options are available only on Windows and UNIX.

- SSQL_CACHE_CHAIN—This option instructs %SSC_RELEASE to cache connections and preserves the cache when chaining to other programs.
- SSQL_CACHE_CONNECTION—This option also instructs %SSC_RELEASE to cache connections, but preserves the cache only for the life of the current program. Cached connections are closed when chaining to other programs.

These options can be overridden by passing *force_release* as a non-zero value:

- If *force_release* is passed as non-zero, %SSC_RELEASE closes connections, regardless of SSQL_CACHE_CHAIN or SSQL_CACHE_CONNECTION settings.
- If *force_release* is not passed or is passed as zero, %SSC_RELEASE caches connections if either option is in force.

%SSC_RELEASE should be the last function call for a database channel unless you want to keep the connection open (not cached) across a chain of applications. In this case, use %SSC_CMD and set the SSQL_KEEP_OPEN option, which is available only on Windows and UNIX.

After a call to %SSC_RELEASE, a call to %SSC_INIT must be issued prior to a call to %SSC_CONNECT. Every call to %SSC_CONNECT should be paired with calls to %SSC_RELEASE and %SSC_INIT.

Examples

The following code segment demonstrates how to release connections when multiple connections have been made.

```
if (%ssc_connect(dbchn, user))
  goto err_exit
if (%ssc_connect(dbchn1, user))
  goto err_exit
.
.
.
if (%ssc_release(dbchn1))
  begin
    sts = %ssc_getemsg(dbchn1, msg, len)
    if (len)
      writes(1, "DB1: " + msg(1,len))
    else
      writes(1, "DB1: Release failed - No error message available.")
    sts = %ssc_release(dbchn1)
  end
.
.
.
if (%ssc_release(dbchn))
  begin
    sts = %ssc_getemsg(dbchn, msg, len)
    if (len)
      writes(1, "DB0:" + msg(1,len))
    else
      writes(1, "DB0: Release failed - No error message available.")
    sts = %ssc_release(dbchn)
  end
```

%SSC_ROLLBACK – Roll back a transaction

WT	WN	U	V

value = %SSC_ROLLBACK(*dbchannel[, mode]*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

SSQL_FAILURE Failure

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

mode

(optional) Indicates whether to roll back and/or start a transaction. (The default differs from one database to another, and *mode* is ignored for databases that don't support different transaction states.) (**n**)

SSQL_TXOFF Roll back the transaction for *dbchannel*.

SSQL_TXON Roll back current transaction for *dbchannel* (if there is one) and start a new explicit transaction.

Discussion

Depending on *mode*, %SSC_ROLLBACK either rolls back a transaction, starts a new transaction, or both, If it rolls back a transaction, it hard-closes any associated cursors.

- If you pass SSQL_TXOFF, %SSC_ROLLBACK rolls back the transaction for *dbchannel*.
- If you pass SSQL_TXON, and your database supports explicit transactions, %SSC_ROLLBACK starts a new transaction for *dbchannel*. If a transaction already exists for *dbchannel*, %SSC_ROLLBACK first rolls back that transaction and then starts the new one.

For databases, such as SQL Server, that support both explicit and implicit transactions, %SSC_ROLLBACK starts an explicit transaction. However, for Oracle and other databases that don't support explicit transactions, %SSC_ROLLBACK does not start a transaction. In this case,

- the first data access (DML) operation (%SSC_OPEN, %SSC_EXECECUTE, %SSC_EXECIO) starts an implicit transaction.
- %SSC_ROLLBACK rolls back the transaction, and SSQL_TXON has the same affect as SSQL_TXOFF.

Synergy databases do not support rollbacks. However, this call will still change the transaction mode.

For more information, see "Transactions and Autocommit" on page 2-49.

Examples

The following example shows how to roll back or commit a transaction, depending on the user's selection.

```
if (abandon) then
  if (%ssc_rollback(dbchn, SSQL_TXOFF))
    goto err_exit
else
  if (%ssc_commit(dbchn, SSQL_TXOFF))
    goto err_exit
```

%SSC_SCLOSE – Soft close one or more open cursors

WT	WN	U	V

value = %SSC_SCLOSE(*dbchannel, dbcursor[, ...]*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

dbcursor

One or more logical cursor numbers within the range from 1 through the maximum number specified by *maxcur* during %SSC_INIT. These cursors must have been opened by %SSC_OPEN. (**n**)

Discussion

%SSC_SCLOSE soft closes one or more logical cursors opened in %SSC_OPEN. A soft close enables SQL Connection to reuse the associated database cursor, if there is one. For information on closing and reusing cursors, see "Closing cursors" on page 2-27 and "Reusing cursors" on page 2-28. Note the following:

- Use %SSC_SCLOSE only when you will reuse the cursor. Otherwise, use %SSC_CLOSE to free *dbcursor* and its resources.
- A table cannot be dropped (that is, deleted) unless all cursors are hard closed.
- If you've specified fewer database cursors than logical cursors (with %SSC_INIT), SQL Connection may hard close cursors that have been soft closed with %SSC_SCLOSE. However, when a cursor is reused with %SSC_OPEN, the result is the same as if the cursor had not been hard closed. In this way, SQL Connection manages a cache of cursors for you.
- For SQL Server, %SSC_SCLOSE frees cursor resources, including locks, for cursors for SELECT statements.
- The database cache may reach its limit, which will result in a severe decrease in performance if you do not hard close cursors.
- If you use ^VARARGARRAY, note that *dbcursor* is the last declared argument for this routine.

Examples

The following code segment shows re-use of a soft closed cursor (the SELECT statement isn't re-parsed).

```
sqlp = "SELECT name, id, type FROM objects "
  &      " WHERE name = :1 AND type = :2"
do forever
  begin
    call get_name_and_type        ;Set the search name and type
    if (sts = %ssc_open(dbchn, cur1, sqlp, SSQL_SELECT,
  &                     SSQL_STANDARD, 2, spec, stype))
      goto err_exit
    if (%ssc_define(dbchn, cur1, 3, name, id, type))
      goto err_exit
    sts = %ssc_move(dbchn, cur1, 1)
    if (sts.eq.SSQL_NOMORE) then
      exitloop
    else if (sts.eq.SSQL_FAILURE)
      goto err_exit
    call do_processing
    if (sts = %ssc_slose(dbchn, cur1))
      goto err_exit
  end
```

%SSC_SQLLINK – Link a non-SELECT statement to cursor for a SELECT statement

WT	WN	U	V

value = %SSC_SQLLINK(*dbchannel, dbcursor, statement[, numvars][, var, ...]*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

dbcursor

The logical cursor number within the range from 1 through the maximum number specified by *maxcur* during %SSC_INIT. The cursor must have been opened by %SSC_OPEN. (**n**)

statement

The SQL statement. (**a**)

numvars

(optional) The number of bound host variables that follow. Note that *numvars* cannot exceed the value of *maxcol* passed in the %SSC_INIT call. (**n**)

var

(optional) Host variables to be bound to the non-SELECT statement. If *numvars* is passed, the number of *var* arguments passed must equal the value of *numvars*. (**n**)

Discussion

%SSC_SQLLINK links a new non-SELECT statement to an already opened SELECT statement cursor, clears all defined variables and bound variables, and rebinds variables for the new non-SELECT statement to follow. The primary use of %SSC_SQLLINK is to update the current row just fetched through %SSC_MOVE (with the SSQL_POSITION and SSQL_FORUPDATE options set in the %SSC_OPEN for the cursor).

Note the following:

- Note that you can call %SSC_MOVE and %SSC_EXECUTE multiple times for the same set of %SSC_OPEN and %SSC_SQLLINK statements, which enables you to repeatedly fetch and update rows using the same cursor.
- To bind more than 250 variables for a statement specified by %SSC_SQLLINK, put up to 250 variables in the %SSC_SQLLINK call, then put the remainder in one or more %SSC_BIND calls.
- Don't put a restriction clause in the update statement passed to %SSC_SQLLINK. When you specify SSQL_POSITION and SSQL_FORUPDATE in the %SSC_OPEN call for the cursor, SQL Connection automatically positions the cursor so restrictions aren't necessary for the update statement.
- If you use ^VARARGARRAY, note that *numvars* is the last declared argument for this routine.

For more information on binding host variables, see "Binding data" on page 2-34.

Examples

For an example, see **exam_fetch_update.dbl**, which is in the connect\synsqlx subdirectory of the main Synergy/DE installation directory.

%SSC_STRDEF – Define a structure

WT	WN	U	V

value = %SSC_STRDEF(*dbchannel, dbcursor, element_num, layout_def, rec*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

dbcursor

The logical cursor number. This must be between 1 and the maximum number specified by *maxcur* during %SSC_INIT (inclusive). The cursor must have been opened by %SSC_OPEN. (**n**)

element_num

The number of array elements in each field of *rec*. (When using *element_num*, be aware that not all databases support bulk insertions or allow multirow operations. For more information, see %SSC_INIT on page 3-35.) (**n**)

layout_def

A record that describes the layout of *rec*. (**a**)

rec

The record SQL Connection will use at execution time to bind or define data. The *layout_def* structure describes the layout for this record. (**a**)

Discussion

%SSC_STRDEF provides an alternative way to bind and define data (see "Data Mapping" on page 2-33). It's generally best to use %SSC_OPEN, %SSC_BIND, %SSC_SQLLINK, or %SSC_DEFINE instead. (For the most part, %SSC_STRDEF has been superseded by support for real arrays in these and other SQL Connection functions.) However, use %SSC_STRDEF if

- you need more than 248 variables for a SELECT statement or more than 256 bind variables for a non-SELECT statement. (See "Binding data" on page 2-34.)
- you are passing many variables to %SSC_DEFINE, %SSC_OPEN, or %SSC_SQLLINK. In this case, using %SSC_STRDEF may improve performance.

Note that if %SSC_STRDEF follows an %SSC_SQLLINK call, %SSC_STRDEF binds variables only for the original SELECT statement (not the linked statement).

Examples

```
.define ELMNT_NUM 5
.include "ssql.def"

record ar_data                          ;Data record structure
      s_dnum       ,[ELMNT_NUM]i4
      s_dnam       ,[ELMNT_NUM]a6
      s_dman       ,[ELMNT_NUM]d4
      s_ddiv       ,[ELMNT_NUM]a10

static record layout_def
      snm_vars       ,d3              ;Number of variables
      group ssql_vars ,[4]a           ;Array of field
                                      ; definitions
        sfld_typ     ,a1              ;Field type (A/D/I)
        sfld_siz     ,d5              ;Field length
        sfld_dec     ,d2              ;Field decimal point length
      endgroup

record row_count, i4

proc
.
.
.
;Build the structure definition
if.NOT. snm_vars
  begin
    snm_vars = 4
    ssql_vars[1].sfld_typ = 'I'
    ssql_vars[1].sfld_siz = 4
    ssql_vars[1].sfld_dec = 0
    ssql_vars[2].sfld_typ = 'A'
```

```
    ssql_vars[2].sfld_siz = 6
    ssql_vars[2].sfld_dec = 0
    ssql_vars[3].sfld_typ = 'D'
    ssql_vars[3].sfld_siz = 4
    ssql_vars[3].sfld_dec = 0
    ssql_vars[4].sfld_typ = 'A'
    ssql_vars[4].sfld_siz = 10
    ssql_vars[4].sfld_dec = 0
  end

;Set the SQL statement
sqlp = "SELECT deptnum, deptname, manager, division FROM org"
if (%ssc_open(dbchannel, cur3, sqlp, SSQL_SELECT))
  goto err_exit

;Define the structure for ELMNT_NUM elements in each array
if (%ssc_strdef(dbchannel, cur3, ELMNT_NUM, layout_def, ar_data))
  goto err_exit

;Fetch the five rows at once
sts = %ssc_move(dbchannel, cur3, ELMNT_NUM, row_count)
for ix from 1 thru row_count
  begin
    anum = s_dnum[ix]
    writes(1, anum + s_dnam[ix] + %a(s_dman[ix]) + s_ddiv[ix])
  end
if (%ssc_close(dbchannel, cur3))
  goto err_exit.
```

4

Utility Functions

SQL Connection utility functions enable you to get information, map error codes, and set date and time options during the execution of your Synergy application. Each utility function returns a value and can be used any place a literal can be used in a Synergy program.

%SSC_GETDBID – Get database ID

WT	WN	U	V

value = %SSC_GETDBID(*dbchannel, dbid*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

SSQL_FAILURE Failure

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT. (**n**)

dbid

The returned database ID. (**n**)

Discussion

%SSC_GETDBID gets the database ID for the specified database channel. The following table lists the possible database IDs.

Database IDs		
Database system ID	**ID#**	**Database**
SSQL_DID_NONE	n/a	*Not connected (or released by %SSC_RELEASE)*
SSQL_DID_ORACLE	0	Oracle
SSQL_DID_RDB	1	Oracle Rdb[a]
SSQL_DID_SYBASE	2	Sybase[a]
SSQL_DID_SDAPI	4	Synergy Database

Database IDs (Continued)		
Database system ID	**ID#**	**Database**
SSQL_DID_INFORMIX	5	Informix[a]
SSQL_DID_OLEDB	8	OLE DB-compliant[a]
SSQL_DID_ODBC	11	ODBC-compliant[a]
SSQL_DID_SQLSRV	12	SQL Server
SSQL_DID_MYSQL	14	MySQL

a. Support for these databases may require assistance from Synergex Professional Services and additional support fees. Contact your Synergy/DE account manager for details. See "Synergex Professional Services Group" on page ix.

Examples

The following example demonstrates how to use %SSC_GETDBID:

```
user = "sa/manager"              ;Default database connection
if (%ssc_connect(dbchn, user))
  goto err_exit
if (%ssc_getdbid(dbchn, dbid))   ;Get the database ID
  goto err_exit
if (dbid.eq.SSQL_DID_SQLSRV)
  if (%ssc_cmd(dbchn, cur1, SSQL_USEDB, "Synergex"))
    goto err_exit                ;Cur1 ignored
```

%SSC_GETEMSG – Get database error message

WT	WN	U	V

value = %SSC_GETEMSG(*dbchannel, msg, len, [row_count][, errno]*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

SSQL_FAILURE Failure

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT. This argument has a maximum size of 1024. (**a**)

msg

Returned with the extended error message. (**a**)

len

Returned with the length of the error message. (**n**)

row_count

(optional) Returned count of rows affected in the last call to %SSC_EXECUTE, %SSC_MOVE, or %SSC_EXECIO. This count is valid only when *value* is returned as SSQL_NORMAL. (**n**)

errno

(optional) Returned with the last database-specific error number. For Synergy databases, this is the Synergy Database API error number. (**n**)

If this number is negative, it's probably a Vortex API error message (see "Vortex API error messages" on page 5-15).

Discussion

%SSC_GETEMSG returns Vortex API, SQL OpenNet, socket, database errors, and if you're using an ODBC database driver (VTX11, VTX12_ODBC, or VTX12_SQLNATIVE), ODBC Driver Manager warnings and errors. (It doesn't, however, return Synergy runtime errors. You'll need to trap Synergy runtime errors in your program—see "Trapping runtime errors" in the "Error Messages" chapter of *Synergy Tools*—and you'll need to use Vortex API logging to view socket errors.)

Some databases return multiple message lines for %SSC_GETEMSG. These lines are separated by the null character (%char(0)). For an example of message decoding, see **printmsg.dbl**. (You'll find this file in the connect\synsqlx subdirectory of the main Synergy/DE installation directory.)

Examples

The following code segment displays a connection failure error message.

```
      if (%ssc_connect(dbchn, user))
        goto err_exit
      .
      .
      .
err_exit,
      sts = %ssc_getemsg(dbchn, msg, len)
      if (len)
        writes(1, msg(1,len))
      else
        writes(1, "No error message available.")
```

%SSC_MAPMSG – Map a database-specific error code

WT	WN	U	V

value = %SSC_MAPMSG(*dbchannel, mapfile, dfltcode, mapcode*)

Return value

value

This function returns an integer result. (**i**)

SSQL_NORMAL Success

SSQL_FAILURE Failure

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

mapfile

The filename (without filename extension) that contains the error code mapping. (**a**)

dfltcode

The code to return if no match is found in the map file. (**n**)

mapcode

Returned with the mapped code. (**n**)

Discussion

%SSC_MAPMSG maps a database-specific error code to a generic error code.

The text file that *mapfile* is set to must contain two numbers in each line: the first number is the database-specific error code (most are negative), and the second number is the matching generic error code that you want returned. SQL Connection opens the error code mapping file once for each open database channel and appends the *xx* filename extension to the name of the *mapfile* text file, where *xx* is the database ID from the last %SSC_GETDBID function call.

When a status value of SSQL_FAILURE is returned, you can call this function to get the map message code. You can then process general error messages in your own error processing system.

Examples

The following code segment demonstrates how to map a database-specific error code to an application generic error code using an Oracle database (with a *mapfile* of **my_map.0**).

```
mapfile = "my_map"
sts = %ssc_getemsg(dbchn, msg, len,, ecode)
sts = %ssc_mapmsg(dbchn, mapfile, ecode, map_code)
call do_case_map_code
```

The contents of the first line of the file if it mapped Oracle error –2 to a returned error 3 would be

```
-2 3
```

%SSC_OPTION – Set or get date and time options

WT	WN	U	V

value = %SSC_OPTION(*dbchannel, mode, base_date, format_string, null_mask*)

Return value

value

This function returns an integer result: (**i**)

SSQL_NORMAL Success

SSQL_FAILURE Failure

Arguments

dbchannel

An internal database channel previously initialized using %SSC_INIT and connected by %SSC_CONNECT. (**n**)

mode

One of the following modes: (**n**)

SSQL_SETOPT Set the following options.

SSQL_GETOPT Get the following options.

base_date

Returns or sets a value that's used to adjust numeric dates (decimal or integer) fetched from the database. We recommend leaving *base_date* set to its default, which is -1721378. See the Discussion below for instructions. (**n**)

format_string

Returns or sets the date/time format string. The maximum number of characters is 64. The default mask is DD-MON-YYYY. (**a**)

null_mask

Returns or sets a value that is bit OR'd to the field when a column is described and nulls are allowed. This is for internal use only. Leave this option set to its default value, which is 0. (**n**)

Discussion

%SSC_OPTION either gets current date and time option settings or sets date and time options. To get the current date and time option settings, use SSQL_GETOPT.

To set an option, do the following:

1. Call %SSC_OPTION using SSQL_GETOPT to retrieve current settings into variables.
2. Assign new values to the variables whose settings you want to change.
3. Call %SSC_OPTION using SSQL_SETOPT. This will update the setting for any variable whose value you changed in step 2, but will leave other options with their original settings.

Base_date determines the base date for fetched dates. When a date is fetched into a numeric output variable, the date is translated into a Julian date, which is a value that represents the number of days between the returned date and the beginning of AD 1. The value of *base_date* is then added to or (if it's negative) subtracted from the Julian date. (A positive value moves the base date forward to a later AD year. A negative value moves the base date back to a BC year.) We recommend that you leave *base_date* set to its default (-1721378), which is compatible with the Synergy DBL routines %NDATE and %JPERIOD.

The following table lists the formatting options for date/time data. The width of the resulting data is determined by the length of the mask.

Date/Time Formatting Characters	
Sequence	**Description**
YYYY	Four-digit year
YY	Two-digit year
RR	Two-digit year from another century—this is a sliding window format based on 20
MM	Two-digit month of year (01-12)
MON	Three-character month (all uppercase)
mon	Three-character month (all lowercase)
Mon	Three-character month (1st character uppercase)
MONTH	Fully named month (all uppercase)
month	Fully named month (all lowercase)
Month	Fully named month (1st character uppercase)
DDD	Three-digit day of year (001-366)

Date/Time Formatting Characters (Continued)	
Sequence	**Description**
DD	Two-digit day of month (01-31)
D	Single-digit[a] day of week (1-7)[b]
DY	Three-character day (all uppercase)
dy	Three-character day (all lowercase)
Dy	Three-character day (1st character uppercase)
DAY	Fully named day (all uppercase)
day	Fully named day (all lowercase)
Day	Fully named day (1st character uppercase)
HH12	Two-digit hour (00-11)
HH,HH24	Two-digit hour (00-23)
MI	Two-digit minutes (00-59)
SS	Two-digit seconds (00-59)
SSSSS	Seconds past midnight (00000-86399)
J	Julian day[a]
Q	Single-digit[a] quarter of year (0-4)
UUUUUU	Microsecond (datetime only)
W	Single-digit[a] week of the month (1-4)[b]
WW	Two-digit week of the year (01-52)
other	Delimiting character: slash (/), dash (-), etc.

a. A single-character mask will not work if it is the only character in a format string. It will work if there are other characters (mask characters and/or non-mask characters).
b. Weeks start with Sunday.

Note the following:

- Adding **th** (which *not* case sensitive) to any uppercase digit mask appends **ST**, **ND**, **RD**, or **TH**, as appropriate, to the date string at the indicated place. For lowercase digit masks, lowercase letters are appended.
- When embedding characters in a string that's part of the mask, enclose in double quotes any characters that are valid masks. For example, if you want the word "day" as part of the format, enclose it in double quotes as in the following: 'DDDth "day"'.

The following table lists some example date/time values, some masks that could be applied to those values, and the results.

Retrieved date	Mask	Result
Feb 6, 1958	`"DD/MM/YYYY"`	"06/02/1958"
Feb 6, 1958	`"qth quarter of YY"`	"1st quarter of 58"
Feb 6, 1958	`"YYYYMMDD"`	"19580206"
Nov 1, 1995 20:48:46	`"HH12:MI on Day"`	"08:48 on Wednesday"
Dec 1, 1994	`'DDDth "day"'`	335TH day

Examples

The following example changes the date format mask. Note that %SSC_OPTION is called twice, as recommended in the the Discussion above.

```
sts = %ssc_option(dbchn, SSQL_GETOPT, date_base, date_fmt, null_mask)
date_fmt = "MM-DD-YYYY"
sts = %ssc_option(dbchn, SSQL_SETOPT, date_base, date_fmt, null_mask)
```

5

Error Logging and Messages

Troubleshooting and Error Logging

The first step in troubleshooting is to make sure Connectivity Series is configured correctly and that your SQL Connection application can successfully connect to the database. (For client/server configurations, this means that you must be able to connect to the SQL OpenNet server.) The following utilities and logging options help you do this:

- **dltest** — This utility lists needed Connectivity Series DLLs and states whether SQL Connection can find them. To use **dltest**, run it from the command line. It is in the synergyde\connect directory and has no options or arguments.
- **vtxping** and **synxfpng** — These enable you to ping an SQL OpenNet server so you can verify that you can connect in a client/server configuration. **Vtxping** and **synxfpng** (when used with the **-x** option) are nearly identical, but **synxfpng** has a verbose option (**-v**) that lists socket calls as they succeed or fail, which can be useful when debugging. For more information, see "The vtxping Utility" in the "Configuring Connectivity Series" chapter of the *Installation Configuration Guide* and "The synxfpng Utility" in the "Configuring xfServer" chapter of *Installation Configuration Guide*.
- **vtxnetd/ vtxnet2** logging — If you set the **log** option for either of these, a log file (**tcm_*pid*.log**) records connection requests and, if the program can't start a worker thread or process, logs the reason for the failure. You may be able to use this to determine why a connection fails in a client/server configuration. This log also records ping and kill requests. See "The vtxnetd and vtxnet2 Programs" in the Configuring Connectivity Series" chapter of the *Installation Configuration Guide*.
- SSQLLOG — This environment variable (part of SQL Connection logging; see "SQL Connection logging" on page 5-5) enables you to see the connect string (with the password masked by asterisks) sent to the database when a connection fails. It works for both client/server and stand-alone configurations.

If Connectivity Series appears to be configured correctly, but you are still unable to connect,

- check encryption settings on the client (in **net.ini**) and on the server (in the **vtxnetd** or **vtxnet2** command line). Make sure these match (or try connecting after removing these settings from both locations), and make sure **net.ini** is in the directory specified by VORTEX_HOME. (Note that if you have both 32-bit and 64-bit Connectivity Series on a 64-bit Windows machine, VORTEX_HOME is set by the last version installed.) See "Using network initialization files to set network defaults" on page 1-7 for more information, and note that mismatched encryption settings or the inability to access **net.ini** encryption settings, can cause a variety of errors when connecting. These include invalid user name, null password, invalid connect string syntax, and data source name errors.
- for SQL Server, make sure you are using SQL Server authentication (not just Windows authentication, which SQL Connection can't use). For example, for SQL Server 2008, make sure the "SQL Server and Windows Authentication mode" option is selected in the Security section of Server Properties.

In addition to the logging options listed above, the Connectivity Series installation automatically sets the environment variable VORTEX_HOST_SYSLOG, which instructs the SQL OpenNet server to generate messages for the event log (Windows), **syslog** (UNIX), or the operator console (OpenVMS) when an attempt to connect to an SQL OpenNet server causes fatal errors. We don't recommend changing this setting. For more information, see VORTEX_HOST_SYSLOG in the "Environment Variables" chapter of *Environment Variables & System Options*.

Once you can connect…

Once you know that your SQL Connection application can connect to the database, you can use the types of logging discussed in this chapter: SQL Connection logging, Vortex API logging, Vortex host logging, and database-specific logging. Figure 5-1 on page 5-4 illustrates where these types of logging apply once the program has connected to the database. (With the exception of the SSQLLOG environment variable mentioned above, the types of logging discussed in this chapter are useful *only* when your SQL Connection application has successfully connected to the database.) In general, because networks and OpenNet Server complicate matters, it's best to start by using logging in a stand-alone configuration. Then, when your program works smoothly in a stand-alone configuration, move to a client/server configuration.

- SQL Connection logging — Records cursor status and SQL Connection API calls. Use this to find SQL Connection API calls in your code that behave differently than expected. For more information, see "SQL Connection logging" on page 5-5.
- Vortex API and Vortex host logging — Record SQL commands and SQL Connection internal information. You can use these to see how an SQL statement is broken down into commands, and you can use these for performance tuning. For more information, see "Vortex API logging" on page 5-6 and "Vortex host logging" on page 5-8.
- Database-specific logging — Use database-specific logging to examine database-specific errors. For information on database-specific logging for Synergy databases, see "Synergy DBMS logging (Synergy database driver only)" on page 5-9.

We also recommend the following:

- Check the return value of all SQL Connection function calls, use %SSC_GETEMSG to retrieve error messages (see %SSC_GETEMSG on page 4-4), and process all error codes as necessary.
- Use bounds checking (**-qcheck** or **/CHECK_BOUNDS**), which instructs the runtime to report errors if your program subscripts outside the bounds of a field. Additionally, for SQL Connection, bounds checking makes sure that stack records are not used for %SSC_ function operations, which would lead to memory corruption.

If you can't solve a problem by examining the log files, save the log files and call Synergy/DE Developer Support. Support will also need a description of the problem and the version numbers of all relevant software and hardware—especially the Synergy/DE version, operating system, database, and database version.

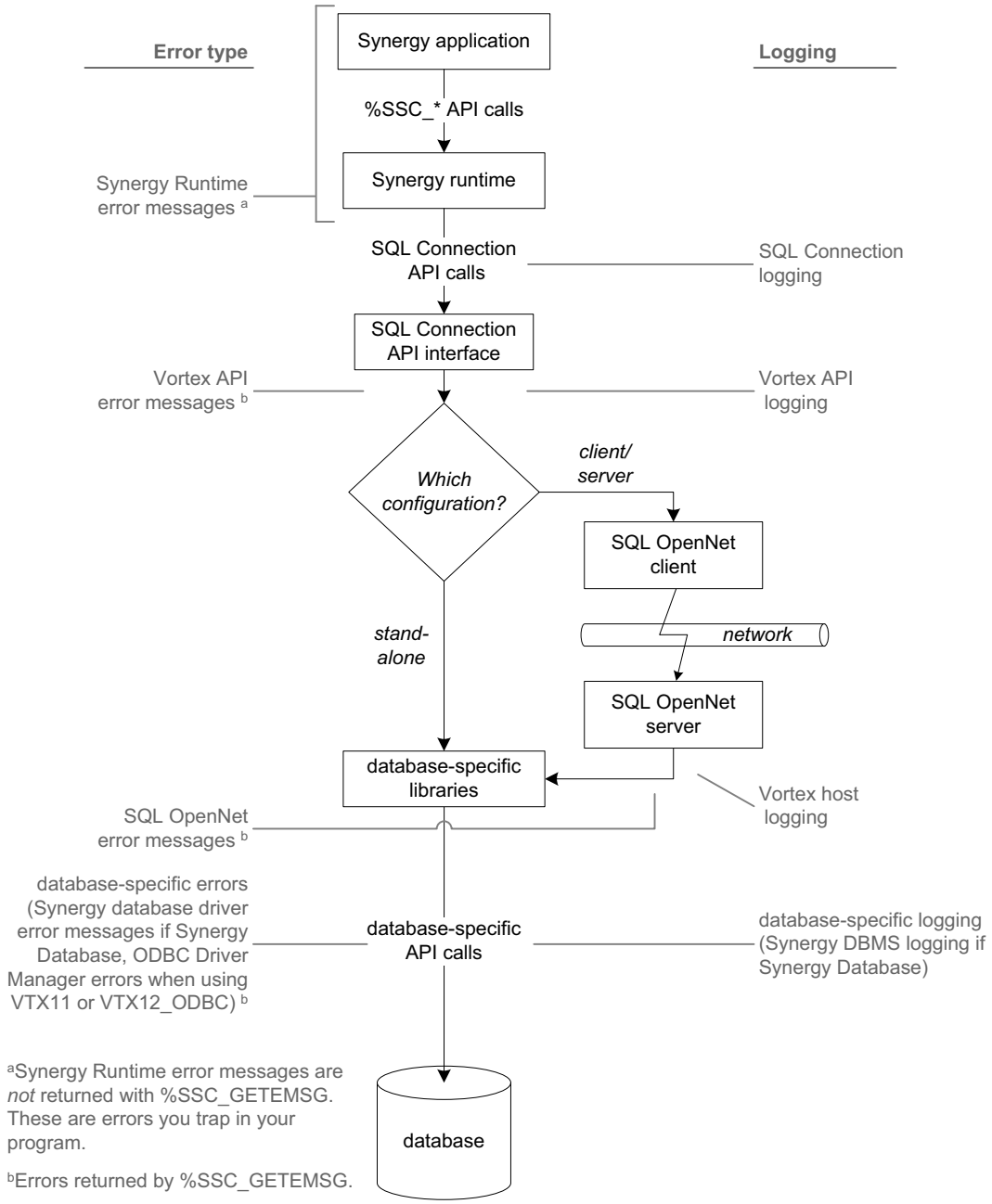


Figure 5-1. Errors and logging for a program that has successfully connected.

SQL Connection logging

SQL Connection logging enables you to track SQL Connection API operations, see the connection string used when a connection fails, list open cursors, and create a detailed log for use by Synergy/DE Developer Support.

To use SQL Connection logging, set one of these environment variables in **synergy.ini** (Windows only) or in the environment. For client/server configurations, set them on the client.

Set...	To...	Explanation
SSQLLOG	1	Creates the **SSQLX.LOG** log file in your working directory, which lists ▸ SQL Connection API operations in the order they were sent to the SQL Connection API interface. ▸ %SSC_*xxx* function calls as well as errors and warnings (in certain circumstances). ▸ open cursors if one of the following errors occurs: $ERR_CURSERR ("ID not select error" or "ID must be a non select cursor") or $ERR_NOMORECURS ("No more available open cursors"). ▸ the connect string (with the password masked by asterisks) that is sent to the database when a connection fails. This log also indicates whether an %SSC_OPEN call reused a cursor, closed a cursor and then reopened it, reopened a closed cursor, or created a new cursor.
SQLJUSTINTIME	1	Records cursor status on error condition. Helps you determine why an operation fails by creating the log file **ssqlerr.log** (in your working directory), which contains an open cursor listing. If **ssqlerr.log** already exists, new logging is appended to the current file, potentially creating a very large file. Note that you cannot use this if SSQLLOG is set.

Vortex API logging

Vortex API logging enables you to see the exact SQL commands passed to the SQL OpenNet client (in a client/server configuration) or to the database driver (in a stand-alone configuration). You can use this information to debug SQL statements, and you can use it to verify optimization. (See "Using Vortex API logging to verify optimization" below.)

To use Vortex API logging, set one or more of the following environment variables in **net.ini** (Windows only) or in the environment. Set them on the client in a client/server configuration.

Set...	To...	Explanation
VORTEX_API_LOGFILE	*Filespec*	Logs the exact SQL commands passed to the SQL OpenNet client (if client/server) or database driver (if stand-alone). If you set this without setting VORTEX_API_LOGOPTS, a list of operations with a total count for each operation is recorded. Note: Don't specify an extension for the filename (or a version number on OpenVMS). SQL Connection automatically appends the process ID (*filename_pid*) and an extension (**.log**). If you specify an extension on OpenVMS, no log file will be created.
VORTEX_API_LOGOPTS	*Options*	Must be used with VORTEX_API_LOGFILE, and must be set to one or more of the following. To set more than one option, separate options with the plus sign—for example, FULL+TIME. APPEND—Appends logging information to existing file. ERROR—Logs only statements with errors. FULL—Specifies full logging. Note: If your program opens multiple database channels concurrently, you must also set MULTI (or you'll get an error). MULTI—Creates a separate file for each channel when using multiple database channels. PLAY—Enables Synergy/DE Developer Support to playback an operation. RECORD—Logs data for Synergy/DE Developer Support. SQL—Creates a file that contains SQL commands. Specify the filename (minus extension) and path (optional) with VORTEX_API_LOGFILE. The extension is **.sql**. TIME—Logs execution time for statements.

Using Vortex API logging to verify optimization

You can use Vortex API logging to find out how well you've optimized cursor usage. (We suggest you use VORTEX_API_LOGOPTS=TIME.) The final page of the log lists counts of Vortex API calls and indicates which operations reuse cursors.

- OPEN statements that reuse cursors are listed as OPENFETCH statements.
- FETCH statements that reuse cursors are listed as FETCHNEXT statements.
- The names of all other statements that reuse cursors end in 2.

In the following example, the EXEC2 count indicates that 100 EXEC statements reused cursors, and the OPENFETCH count indicates that 200 open statements reused cursors.

```
EXEC                5
EXEC2               100
OPEN                20
OPENFETCH           200
```

For best performance, each statement should have more operations that reuse cursors than operations that don't.

Note that you can also use SSQLLOG (SQL Connection logging) to see if cursors are being opened and closed for a series of identical SELECT statements, where a single cursor with %SSC_SCLOSE (or no close at all) should be used.

Vortex host logging

Like Vortex API logging, Vortex host logging records SQL commands. But Vortex host logging logs these commands as they are passed from the SQL OpenNet server to the database driver. You can use this information to debug SQL statements and to verify optimization. (See "Using Vortex API logging to verify optimization" above for information.)

To use Vortex host logging, set one or more of the environment variables listed in the following table. Set these on the server.

- On Windows, set them in the **opennet.srv** file before starting **vtxnetd** or **vtxnet2**.
- On UNIX, set them in the environment before starting **vtxnetd**.
- On OpenVMS, set them as system-wide logicals before starting the server.

Set…	To…	Explanation
VORTEX_HOST_LOGFILE	*Filespec*	Logs SQL commands as they are passed from SQL OpenNet to the database driver. If you set this without setting VORTEX_HOST_LOGOPTS, a list of operations along with a total count for each operation is recorded. Note: Don't specify an extension for the filename (or a version number on OpenVMS). SQL Connection automatically appends the process ID (*filename_pid*) and an extension (**.log**). If you specify an extension on OpenVMS, no log file will be created.
VORTEX_HOST_LOGOPTS	*Options*	Must be used with VORTEX_HOST_LOGFILE, and must be set to one or more of the following. To set more than one option, separate options with the plus sign—for example, FULL+TIME. APPEND—Appends logging information to existing file. ERROR—logs error statements only. FULL—Specifies full logging. If your program opens multiple database channels concurrently, MULTI must also be set (or you'll get an error). MULTI—Creates a separate file for each channel when using multiple database channels. Set automatically when running **vtxnetd** on Windows. PLAY—Enables Synergy/DE Developer Support to playback an operation. RECORD—logs data for Synergy/DE Developer Support. SQL—Creates a file that contains SQL commands. Specify the filename (minus extension) and path (optional) with VORTEX_HOST_LOGFILE. The extension is **.sql**. TIME—logs execution time for statements.

Synergy DBMS logging (Synergy database driver only)

If you're connecting to a Synergy database driver, Synergy DBMS logging enables you to create a log of Synergy database driver operations. You can also create a detailed log for Synergy/DE Developer Support.

For information on Synergy DBMS logging, see the "Data Access Errors and Error Logging" chapter of the *xfODBC User's Guide*.

Error Messages

The %SSC_GETEMSG function returns Vortex API, SQL OpenNet, socket, database errors, and if you're using an ODBC database driver (VTX11, VTX12_ODBC, or VTX12_SQLNATIVE), it returns ODBC Driver Manager warnings and errors. It doesn't, however, return Synergy runtime errors. You'll need to trap Synergy runtime errors in your program (see "Trapping runtime errors" in the "Error Messages" chapter of *Synergy Tools*), and you'll need to use Vortex API logging to view socket errors.

For information on database-specific errors, see your database documentation. (For Synergy database driver error messages, see "Data Access Errors" in the "Data Access Errors and Error Logging" chapter of the *xfODBC User's Guide*.)

Synergy runtime error messages

You can trap the following runtime errors in your SQL Connection program. To see the associated error text, use SQL Connection logging, or use $ERR_CATCH in conjunction with %ERR_TRACEBACK. The following errors apply to stand-alone configurations and clients in client/server configurations.

Synergy Runtime Error Messages		
Mnemonic	**Error text**	**Explanation**
$ERR_ARGSIZ	Field or array count (*size*) smaller than statement count (*ct*)	You used an array argument with fewer dimensions than the row count, or you specified fewer arguments than the statement defined.
$ERR_AXUNSUP	Unsupported feature	Your program uses an option that is not supported for the database driver. For example, this error could be caused by specifying SSQL_SCROLL_DYNAMIC, an option for %SSC_OPEN, with any database driver other than VTX11, VTX12_ODBC, or VTX12_SQLNATIVE.
$ERR_BADDATATYP	Wrong data type	An argument with an invalid data type has been passed to a function. Check the argument list for the correct data type.
$ERR_CHNUSE	DB channel in use	The *dbchannel* argument passed to %SSC_CONNECT has already been used for an active connection.

Synergy Runtime Error Messages (Continued)		
Mnemonic	**Error text**	**Explanation**
$ERR_CURSERR	ID must be a non select cursor	The cursor ID passed to a non-SELECT processing function (such as %SSC_EXECIO or %SSC_EXECUTE) is not associated with a non-SELECT statement.
	ID not SELECT cursor	The cursor ID passed to a SELECT processing function (such as %SSC_MOVE or %SSC_DEFINE) is not associated with a SELECT statement.
	Invalid cursor ID	The cursor ID is not associated with an active open cursor.
$ERR_NOMEM	Not enough memory (needed *x*, had *y*)	This operation could not be performed with the available memory. This error occurs only after all memory has been reorganized and all unnecessary segments have been freed.
$ERR_NOMORECURS	No more available open cursors	You have attempted to open a cursor when the maximum number of open cursors defined in %SSC_INIT has already been reached.
$ERR_NOOPEN	Invalid DB index channel used	The *dbchannel* argument is not a valid connection to the database. This may be caused by a failure to establish the channel correctly with %SSC_INIT and %SSC_CONNECT, or it may be a problem with the environment. (For example, it may be that database channels were not opened with consecutive numbers starting with 1, a requirement for %SSC_INIT.) To troubleshoot, run the SQL Connection example programs for your database. ▸ If you are able to run the SQL Connection example programs, the problem lies with the way your program initializes and/or connects to database channels. Use the code for the SQL Connection example programs as a guide. ▸ If you get errors when you run the SQL Connection example programs, there may be a problem with your environment. (For example, you may get DLLLOAD errors.)

Synergy Runtime Error Messages (Continued)		
Mnemonic	**Error text**	**Explanation**
$ERR_CURSERR	ID must be a non select cursor	The cursor ID passed to a non-SELECT processing function (such as %SSC_EXECIO or %SSC_EXECUTE) is not associated with a non-SELECT statement.
	ID not SELECT cursor	The cursor ID passed to a SELECT processing function (such as %SSC_MOVE or %SSC_DEFINE) is not associated with a SELECT statement.
	Invalid cursor ID	The cursor ID is not associated with an active open cursor.
$ERR_NOMEM	Not enough memory (needed *x*, had *y*)	This operation could not be performed with the available memory. This error occurs only after all memory has been reorganized and all unnecessary segments have been freed.
$ERR_NOMORECURS	No more available open cursors	You have attempted to open a cursor when the maximum number of open cursors defined in %SSC_INIT has already been reached.
$ERR_NOOPEN	Invalid DB index channel used	The *dbchannel* argument is not a valid connection to the database. This may be caused by a failure to establish the channel correctly with %SSC_INIT and %SSC_CONNECT, or it may be a problem with the environment. (For example, it may be that database channels were not opened with consecutive numbers starting with 1, a requirement for %SSC_INIT.) To troubleshoot, run the SQL Connection example programs for your database. ▸ If you are able to run the SQL Connection example programs, the problem lies with the way your program initializes and/or connects to database channels. Use the code for the SQL Connection example programs as a guide. ▸ If you get errors when you run the SQL Connection example programs, there may be a problem with your environment. (For example, you may get DLLLOAD errors.)

Synergy Runtime Error Messages (Continued)		
Mnemonic	**Error text**	**Explanation**
$ERR_NOSQL	SQL Connection installation error or DBLOPT48 not set	You have called an SQL Connection routine before initializing SQL Connection. On Windows and UNIX systems, SQL Connection is initialized by setting system option #48 using either DBLOPT or %OPTION. On OpenVMS systems, you must call %INIT_SSQL to initialize SQL Connection. See "Initializing SQL Connection" on page 2-5 for more information. Note that on Windows, **ssqlx.dll** must be present when initializing SQL Connection. On UNIX systems, **SSQLX.so** must be present.
$ERR_SQLERR	Initialize Synergy SQL by calling %INIT_SSQL first (OpenVMS)	You have called an SQL Connection routine before initializing SQL Connection. On Windows and UNIX systems, SQL Connection is initialized by setting system option #48 using either DBLOPT or %OPTION. On OpenVMS systems, you must call %INIT_SSQL to initialize SQL Connection. See "Initializing SQL Connection" on page 2-5 for more information. Note that on Windows, **ssqlx.dll** must be present when initializing SQL Connection. On UNIX systems, **SSQLX.so** must be present.
	Initialize Synergy SQL by setting DBLOPT 48 (Windows and UNIX)	
	Synergy SQL ERROR: uninitialized system called	
	Synergy SQL ERROR: Licensing error Demo period expired	Your 14-day demo period has expired. Please contact Synergex or your Synergy/DE supplier for a configuration key.
	Synergy SQL ERROR: Licensing error Maximum users exceeded	The maximum license capacity in the License Manager has been reached. (In other words, the number of log-ins on your system is greater than the licensed number of users.) Either contact your Synergy/DE supplier for another configuration key so you can increase the number of users, or wait until someone logs out.
	Synergy SQL ERROR: Licensing error Product not installed	SQL Connection is not configured correctly. See the "Configuring Connectivity Series" chapter of the *Installation Configuration Guide* for assistance.

Synergy Runtime Error Messages (Continued)		
Mnemonic	**Error text**	**Explanation**
$ERR_WRITLIT	Writing into a literal	You have attempted to change the value of an alpha, decimal, implied-decimal, or integer literal. Generally, this error occurs when you pass a literal or an expression to a function that requires a variable, and the function attempts to modify the argument.
$ERR_WROARG	Not enough arguments	The number of arguments passed to a function is different than the number of arguments needed for the function.
	Number of variables (*var_ct*) does not match SELECT- column count (*col_ct*)	Too few variables were defined when %SSC_MOVE was called. The cursor passed to %SSC_MOVE is associated with a SELECT statement that returns a greater number of columns than the number of variables defined for them.

Vortex API error messages

The %SSC_GETEMSG function returns the following Vortex API error messages. Errors apply to stand-alone configurations and clients in client/server configurations.

Vortex API Error Messages		
Mnemonic	**Error text**	**Explanation**
BADCONV	Data conversion failed	The requested data conversion failed. Check that the requested data is of the appropriate type. For example, this error will occur if you request a character column to be fetched into an integer and the column includes characters that aren't numbers.
BADSQLDA	SQLDA is invalid	This is an internal error. Turn on Vortex API logging, repeat the steps that caused the error, and then call Synergy/DE Developer Support. (For information on Vortex API logging, see "Vortex API logging" on page 5-6.)
BLOBCOL	Invalid BLOB column ID	The column specified in the SELECT command is not a large binary object (BLOB) column. Check your SQL statement.
BLOBFILE	BLOB file operation *operation_name* failed	For an RDBMS that keeps large binary object (BLOB) data in external files, various file operations could fail. Make sure the process owner has write permissions to the directory.
BLOBLEN	BLOB length mismatch	This is an internal error. Turn on Vortex API logging, repeat the steps that caused the error, and then call Synergy/DE Developer Support. (For information on Vortex API logging, see "Vortex API logging" on page 5-6.)
BLOBPROC	Cannot return BLOB data via a stored procedure	To return binary large object (BLOB) or character large object (CLOB) data, use BLOB/CLOB-specific functions. Some relational databases cannot return BLOB or CLOB data through stored procedure calls.
DLLENTRY	Could not find DB driver entry point *entry_point_name* (handle: *handle_name*)	The loaded DLL or shared library does not contain the expected entry point. Typically, this happens when the wrong DLL has been loaded and occurs only on machines that support DLLs. Verify that none of the **vtx*** DLLs or **VTX*** shared objects have been overwritten by other DLLs and that there are no other DLLs in the path with the same name. Finally, make sure that any resources that the DLL needs are available. You can determine which resources are required for a shared library on most UNIX systems with the **ldd** command. (You must use third-party tools to do this on Windows systems.)

Vortex API Error Messages (Continued)		
Mnemonic	**Error text**	**Explanation**
DLLLOAD	Could not load *DLL_name* (errno: *error_number*)	Could not load a needed DLL or shared library. The DLL or shared library may be missing, it may be invalid (incorrectly named or an incorrect version), its file specification may be missing from PATH (on Windows) or from the library path (on UNIX), or the listed DLL or shared library may not be able to access third-party DLLs or shared libraries it needs. See "Troubleshooting DLLLOAD Errors" on page 5-25 for more information.
DRVCMDI	Expected an integer parameter	The %SSC_CMD function expects the command line's first token to be an integer. Correct your code.
DRVCMDP	Invalid parameter	A parameter to the %SSC_CMD function is invalid. Correct your code.
DRVCONF	Driver not configured	This is an internal error that typically indicates that the database driver process (**VTX0_10**, for example) terminated abnormally. Turn on Vortex host logging, repeat the steps that caused the error, collect any relevant information from the Windows event log (if you're on Windows), and call Synergy/DE Developer Support. (For information on Vortex host logging, see "Vortex host logging" on page 5-8.)
DRVMULTI	Driver must be specified when multiple are present	On Windows and UNIX systems, you must specify the driver names when more than one driver is linked. (If only one driver is present, the driver name is optional.) Additionally, for ISAM access, the connect string must contain "sdms:". For example: `genesis:public//sdms:passport` See "Building Connect Strings" on page 2-16 for more information.
DRVNOTF	Driver *driver_name* not found	The connect string specifies a driver that your installation of SQL Connection is not built to support. Make sure the driver in the connect string is correct.
FETCURCLO	Attempting a FETCH from a closed cursor, *cursor_name*	The cursor used for the FETCH has previously been closed. Check your program logic to make sure it doesn't use a cursor that's been closed.
FLIPOVER	Flip buffer overflow	This error occurs if too many parameters are specified. The current limit is approximately 250 parameters. Note that multiple dimensions are not included in this limit.

Vortex API Error Messages (Continued)		
Mnemonic	**Error text**	**Explanation**
INVCUR	Invalid cursor	The cursor has not been initialized. Set the cursor to **-1** before the first call, and do not modify it after subsequent calls.
INVCURPOS	Invalid cursor for positioned EXEC	The cursor must be a valid cursor from a previous %SSC_OPEN call.
INVDATE	Invalid date/time	The format of the date or time data is invalid. Format the date or time correctly. See "Converting dates and times" on page 2-40 for information on formatting dates.
INVDRVVER	DB version mismatch (expected: *driver_name*, found *version_number*)	The version of the database driver is not at the same level as the SQL OpenNet runtime library. This error is most common when SQL OpenNet client/server is being used, but can also occur if an older driver has been linked with a newer runtime library. Make sure the client and the server are running the same version of SQL Connection.
INVNUM	Invalid (internal) number	The data being converted is invalid. Call Synergy/DE Developer Support.
INVUPD	Invalid UPDATE statement	Invalid UPDATE statement for binary large object (BLOB) processing. This is a Sybase-specific error. Verify the syntax of the UPDATE statement.
MANYBIND	Too many bind(host) variables	There are too many bind variables specified for a particular stored procedure or prepared statement. Make sure the number of bind variables matches the number of bind variable positions in the statement.
MANYCCUR	Too many concurrently open cursors (*cursor name*: *cursor number*)	There are too many cursors open. Either allocate more cursors with %SSC_INIT or close cursors you don't need.
MANYCOLS	Too many columns (*number*) returned by query	The query returns more columns than the database channel has been set to accept. Either modify the query to return fewer columns, or use the *maxcol* argument for %SSC_INIT to increase the number of columns that a query can return. The default maximum is 254 columns. See %SSC_INIT on page 3-35 for more information.
MANYLCUR	Too many logical cursors	Too many logical cursors have been requested. Use %SSC_INIT to allocate more.

Vortex API Error Messages (Continued)		
Mnemonic	**Error text**	**Explanation**
NOCONN	Not connected	A connect must be performed before any other operation.
NODRV	No DB driver linked	When generated for an %SSC_CONNECT call, check the connect string. This error indicates that the database driver specified in the connect string is not available. When generated for a call to any other function, this error indicates that there has been an attempt to reuse a channel that has been released with %SSC_RELEASE, so check your code.
NOMEM	Out of memory	This is a fatal error. Either there is no more heap memory available (which is rare), or the heaps have been corrupted. Notify your system administrator.
n/a	Null password given: logon denied	Encryption is set incorrectly on the client or the server, or the **net.ini** file cannot be found. Check the encryption settings and make sure VORTEX_HOME is set to the correct directory. See "Using network initialization files to set network defaults" on page 1-7.
ORAOOPT	oopt() requires two integer parameters	Oracle's oopt() function requires two integer parameters. See Oracle's OCI documentation for information.
POSBROW	Position EXEC requires a 'for browse' cursor	A positioned EXECute (UPDATE or DELETE) requires a previously opened and positioned cursor. Open the cursor in FOR BROWSE mode. This error occurs only in SQL Server and Sybase.
POSEXEC	Position EXEC requires an open cursor	A positioned EXECute (UPDATE or DELETE) requires a previously opened and positioned cursor. Verify your program logic.
UNDESDTY	Unknown DESCRIBEd data type (*name type_value*)	The data type of a described data type is unknown. This may occur if a relational database introduces a new data type while an older database-specific library is being used. Call Synergy/DE Developer Support.
UNSUPFNC	Unsupported function (*function_name: function_number*)	An unsupported database driver function has been encountered. In most cases, this is the result of a call to a routine or operation that is not supported by your version of the driver or database. Call Synergy/DE Developer Support.

SQL OpenNet error messages

The %SSC_GETEMSG function returns the following SQL OpenNet error messages. Errors apply only to clients in client/server configurations.

SQL OpenNet Error Messages		
Mnemonic	**Error text**	**Explanation**
AUTHBAD	Invalid authentication syntax	The connect string syntax in invalid. Make sure the (*[domain/]uid/pwd*) part of the network string follows the host name. Note that this user ID and password are for the host machine, *not* the database. See "Network string (opennet_info) syntax" on page 2-19 for more information.
n/a	Authentication failed	User and password authentication failed on the server. If you're using a Windows server, make sure the user has "log on as batch job" privileges on the server. In addition, make sure the user and password are correct in the connect string and encryption is set to the same value on both the client and the server.
AUTHFAIL	Authentication on *service* failed	You are not authorized to use the requested host service. Make sure the *([domain\]uid/pwd)* part of the network string is correct or contact your system administrator. Note that this user ID and password are for the host machine, *not* the database. See "Network string (opennet_info) syntax" on page 2-19 for more information.
AUTHREQ	Host *host_name* requires authentication	The host you are connecting to requires additional authentication. If the **-a** option is set for **vtxnetd** or **vtxnet2**, the connect string must include a username and password for an account on the machine or an account for a domain that the machine is part of (in addition to any database log-in information). If you've done this, make sure the username and password are correct. See "Network string (opennet_info) syntax" on page 2-19 for more information.
BADINI	'net.ini' file is either missing or invalid	The **net.ini** file is either missing or invalid. Make sure there's a **net.ini** file on the client and that it's valid.

SQL OpenNet Error Messages (Continued)		
Mnemonic	**Error text**	**Explanation**
CONFIG	Expected a CONFIG call	This is an internal error that typically indicates that the database driver process (**VTX0_10**, for example) terminated abnormally. Collect any relevant information from the event log (Windows), **syslog** (UNIX), or the operator console (OpenVMS), and then call Synergy/DE Developer Support. You may be asked to use Vortex host logging and/or Synergy DBMS logging to assist. For information, see "Vortex host logging" on page 5-8 and "Synergy DBMS logging (Synergy database driver only)" on page 5-9.
Connect:errno:		See "Socket errors" on page 5-23.
DLLENTRY	Could not find DB driver entry point	The loaded DLL does not contain the expected entry point. Typically this happens when the wrong DLL has been loaded and occurs only on machines that support DLLs. Call Synergy/DE Developer Support.
DLLLOAD	Could not load DLL	A needed DLL or shared library could not be loaded. The DLL or shared library may be missing, it may be invalid (incorrectly named or an incorrect version), its file specification may be missing from PATH (on Windows) or from the library path (on UNIX), or the listed DLL or shared library may not be able to access third-party DLLs or shared libraries it needs. See "Troubleshooting DLLLOAD Errors" on page 5-25 for more information.
DLLSAFE	Loaded DLL is not thread safe	The loaded DLL is not thread safe. (Not all database drivers are.) To avoid this error, run the single-threaded daemon, **vtxnet2.exe**.

SQL OpenNet Error Messages (Continued)		
Mnemonic	**Error text**	**Explanation**
EXECFAIL	Exec *program_name* failed on host *host_name* errno=*nnn*	The service (*program_name*) specified in the network connection string could not be started. This will occur if the service could not be found, does not have the correct permissions, or is not listed as a valid service. ▸ Make sure the service exists, is listed as a valid service, and that you are connecting with the correct user name and password. ▸ Use vtxnetd/vtxnet2 logging and check the **tcm_*pid*.log** file. See "The vtxnetd and vtxnet2 Programs" in the Configuring Connectivity Series" chapter of the *Installation Configuration Guide* for information. ▸ Check your operating system documentation for information on the error number (*nnn*).
HOSTNOTFOUND	Host *host_name* not found	The network does not recognize *host_name* as a server name. Contact your network administrator or try using **ping**, **vtxping**, or **synxfpng** (with **-x** option) to check for the server. Make sure the spelling of *host_name* is correct.
n/a	Invalid integer	The number specified for the encryption key is invalid or the **net.ini** file is corrupt. Verify that the key is set to an integer value in the correct range, and make sure the **net.ini** file has no control characters.
n/a	Invalid connect string syntax (uid/pwd/datasource)	The syntax of the connect string is invalid or there's a problem with encryption. Verify the syntax of the connect string and make sure both client and server are running versions of Connectivity Series that support encryption (version 8.1 and later). In addition, make sure the encryption setting in **net.ini** (on the client) matches the encryption setting on the server (set with the **vtxnetd**/**vtxnet2 -k** option).
INVHOSTSYN	Invalid host/service name syntax	The syntax for the host or service name is invalid. For information on connect string syntax, see "Building Connect Strings" on page 2-16.

SQL OpenNet Error Messages (Continued)		
Mnemonic	**Error text**	**Explanation**
INVVER	NET version mismatch (host: *host_ver*, client: *client_ver*)	The version of SQL OpenNet on the server is different than the version on the client. For example, because 1000352 is the number for 8.1 versions of SQL OpenNet, and 1000400 is the number for version 8.3 and higher, the following message indicates that a client with version 8.3 or higher is attempting to access a version 8.1 SQL OpenNet server: **NET version mismatch (host: 1000352, client: 1000400)** Make sure both client and server use the same version of Connectivity Series.
KEEPALIVE	Setting SO_KEEPALIVE failed	This indicates that the socket option KEEPALIVE failed or was not set. Call Synergy/DE Developer Support.
LINGER	Setting SO_LINGER failed	This indicates that the socket option LINGER failed or was not set. Call Synergy/DE Developer Support.
NOINTR	Host cannot be interrupted	Your program requested an %SSC_CANCEL operation, but the host you are trying to cancel cannot handle interrupts. Modify your program so that it doesn't call the cancel operation.
NOMEM	Out of memory	This is a fatal error. Either there is no more heap memory available (which is rare), or the heaps have been corrupted. Notify your system administrator.
Recv:errno		See "Socket errors" on page 5-23.
SERVNOTFOUND	Service/Protocol *name* not found	The service or protocol cannot be found. Make sure the port used for the client matches the port used for the SQL OpenNet server.
SOCKET	Socket() failed	SQL Connection is unable to open a socket. The operating system may have run out of file descriptors. Notify your system administrator.
UNDBID	Unknown Database ID	An unknown database ID is specified in **net.ini**. Use the syntax documented in "Setting connect string defaults and encryption in net.ini" on page 1-7.

Socket errors

The following are the most common TCP/IP socket errors when using SQL Connection. See "Troubleshooting Socket Errors" on page 5-27 for information.

Error text	Error number[a]	Explanation
Connection reset by peer	10054 (WSAECONNRESET) on Windows 54 (ECONNRESET) on UNIX and OpenVMS	This error indicates that a connection to the server has been closed. For information, see "Connection reset by peer (10054 or 54)" on page 5-27.
Connection refused	10061 (WSAECONNREFUSED) on Windows 61 (ECONNREFUSED) on UNIX and OpenVMS	This error indicates that the SQL Connection program can't make a connection to the SQL OpenNet server. For information, see "Connection refused (10061 or 61)" on page 5-28.
Unknown Error		If this error occurs on the server and there are no errors on the client, the SQL Connection program should ignore it and terminate normally. If this error occurs on the client, it indicates that although a connection was gracefully closed by the server, the client was not prepared for it.This is generally caused by either a version mismatch or by network latency issues where the final packet sent by the server is not received before the default server socket shutdown is initiated. This could occur, for example, if the initial connect fails with an error. See the event log (Windows), **syslog** (UNIX), or operator console (OpenVMS) for information on the problem.

a. For UNIX and OpenVMS, 54 and 61 are common error numbers, but error numbers vary for different platforms.

ODBC Driver Manager errors (Windows)

The %SSC_GETEMSG function may return the following ODBC Driver Manager error message, which occurs only if you use the VTX11, VTX12_ODBC, or VTX12_SQLNATIVE database drivers. This error applies to stand-alone connections and servers in client/server connections.

Error text	Explanation
Connection is busy with results for another hstmt	Most likely this is caused by an SQL Connection program that uses client-side cursors but has multiple concurrently active result sets. (Client-side cursors support only a single active result set.) Use server-side cursors.
Data Source name not found and no default driver specified	The DSN specified in the connect string doesn't exist (check the spelling), encryption is set incorrectly on the client or the server, or the **net.ini** file cannot be found. Check the encryption settings and make sure VORTEX_HOME is set to the correct directory. See "Using network initialization files to set network defaults" on page 1-7. If you are on a 64-bit Windows machine, it could be that the DSN has not been defined by the right version of the Microsoft ODBC Administrator. For example, if a 32-bit application is accessing a local database on a system with both 32-bit and 64-bit Synergy/DE, the DSN must be created by the 32-bit ODBC Administrator. For more information, see "Adding a user or system DSN" in the "Configuring Data Access" chapter of the *xfODBC User's Guide*.

Troubleshooting DLLLOAD Errors

DLLLOAD errors indicate that one of the Connectivity Series components can't load a needed DLL or shared library. The DLL or shared library may be missing, it may be invalid (incorrectly named or an incorrect version), its file specification may be missing from PATH (on Windows) or from the library path (on UNIX), the listed DLL or shared library may not be able to access third-party DLLs or shared libraries it needs, or if you're on UNIX, the setuid (+s) bit may be set for the Synergy runtime (**dbr**). DLLLOAD errors occur only on machines that support either DLLs or shared libraries, and these errors are generally caused by a problem with the way Connectivity Series is installed or, on UNIX, by a failure to run **setsde** correctly.

To troubleshoot, run the **dltest** utility from the command line. (The **dltest** utility is in the connect directory and has no options or arguments.) This utility indicates whether needed Connectivity Series DLLs or shared libraries can be accessed, and if you're on UNIX, it tells you the name of the library path environment variable (for example, SHLIB_PATH on HP-UX 32-bit or LIBPATH on IBM AIX 32-bit). In addition, note the following:

- Make sure all of the resources that the DLL or shared library needs are available. For example, on Windows if you get a DLLLOAD error for **GDS0.DLL**, it may be that Connectivity Series can't find one of the DLLs required by **GDS0.DLL**. (These include **SDMS22.DLL** and **VTXIPC.DLL**.) If you get a DLLLOAD error for one of the Synergy database drivers, such as **vtx0_10.DLL** (a driver for Oracle on Windows) or **vtx12.DLL** (the default driver for SQL Server on Windows), it may be that the target database isn't set up correctly or that DLLs for the target database are inaccessible.
  - On Windows, you can use the Dependency Walker utility **(depends.exe)** to determine which resources are required for a DLL. You can download Dependency Walker from **http://www.dependencywalker.com**.
  - On most UNIX systems, you can use the **ldd** command to determine which resources are required for a shared library.
- On UNIX, make sure the setuid (+s) bit is not set for the Synergy runtime (**dbr**). The setuid bit prevents the library path environment variable from being used. This will cause DLLLOAD errors, though it won't affect the ability of **dltest** to access needed **.so** files.

- On UNIX, if **dltest** can access all the needed **.so** files (and the setuid bit is not the problem), there are a few possible causes:
  - For SQL OpenNet, it may be that **setsde** isn't run before **vtxnetd** attempts to implement the SQL OpenNet server. Check the rc file and make sure **setsde** is run before **startnet**.
  - For an SQL Connection client application, it may be that **setsde** isn't run for the process before the application is started. For example, if you run an SQL Connection application from a UNIX processor, such as "cron" or "at", be sure to run **setsde** from the shell script you use to execute your program. The **setsde** script must be run before the SQL Connection application.
  - It may be that the target database isn't set up correctly (in other words, that the database equivalent of **setsde** hasn't been run) or that shared libraries for the target database are inaccessible. On most UNIX systems, you can use the **ldd** command to determine which resources are required for a shared library.

  For information on which Synergy shared libraries are causing the error, run **dltest** from the rc file (directly before the **startnet** command) or from the UNIX command ("cron", "at", etc.) that runs the SQL Connection client application. Remember that the problem may be that Connectivity Series cannot access a third-party shared library needed by the shared library listed in the error (for example, **clntsh.so** may be inaccessible to **VTX0.so** when using Oracle).

Troubleshooting Socket Errors

TCP/IP socket errors have the following mnemonics:

- Connect:errno:*error*
- Recv:errno:*error*

where *error* is the text for the socket error.

To view socket errors, use Vortex API logging or %SSC_GETEMSG. On UNIX systems, see **/usr/include/errno.h**. On Windows systems, you can find these errors in **winsock.h** or **winsock2.h**, which are typically distributed with Microsoft's Platform Software Development Kit (SDK). For details on Microsoft error codes, see Microsoft documentation.

Connection reset by peer (10054 or 54)

The "Connection reset by peer" socket error, which is 10054 (WSAECONNRESET) on Windows and generally 54 (ECONNRESET) on UNIX and OpenVMS, indicates that a connection to the server has been closed. This could be caused by a fatal error on the server, the server stopping, a network problem, or even a connection problem.

1. If the error has the form "connect:errorno:*error*", use **vtxping** (or **synxfpng** with the **-x** option) to test your ability to connect to the server. Otherwise, skip to step 2. The **vtxping** and **synxfpng** utilities print reports to the screen. This information can be used by your network administrator to resolve TCP/IP network socket communication problems.

   - If you can connect, then the network, the server, and the Synergy/DE OpenNet Server service (**SynSQL**) are working. Continue with step 2.
   - If you can't connect, make sure that the server is running, that the **SynSQL** service is running on the server, and that **vtxping** or **synxfpng** is using the correct port.

   For information on **vtxping**, see "The vtxping Utility" in the "Configuring Connectivity Series" chapter of the *Installation Configuration Guide*. For information on **synxfpng**, see "The synxfpng Utility" in the "Configuring *xf*Server" chapter of the *Installation Configuration Guide*.

2. Check and correct the following, which may solve the problem if it is caused by network timing issues:

   - If you're using network licensing, make sure License Manager is configured as a network server.
   - Make sure the server was rebooted after the Connectivity Series components were installed.
   - Check the system-level PATH on the server. It should include the connect directory.
   - Use **vtxnetd** or **vtxnet2** logging and check the resulting **tcm_*pid*.log** file. Then check the event log on Windows, **syslog** on UNIX, or the operator console on OpenVMS. (For information on **vtxnetd** and **vtxnet2** logging, see "The vtxnetd and vtxnet2 Programs" in the "Configuring Connectivity Series" chapter of the *Installation Configuration Guide*.)

- Make sure the file(s) for the database driver you are using are in the connect directory on the server. On Windows, there are two files: an **.exe** and a **.dll** (e.g., **vtx0_10.exe** and **vtx0_10.dll** for Oracle). On UNIX, this is an **.so** file (e.g., **vtx0.so**). On OpenVMS, this is an **.exe** file (e.g., **vtx0.exe**).
- Make sure there is no more than one **vtxnet2** process running on the server. If there's more than one, use **vtxkill** to kill the processes; then restart the service.

3. If you're on Windows or UNIX, run the **dltest** utility to make sure Connectivity Series DLLs or shared libraries are loading properly. (The **dltest** utility is a command line utility in the synergyde\connect directory and has no options.)
4. Make sure the connect string in the client application is referencing a valid driver and database.
5. If you get this socket error again, use **vtxnetd** or **vtxnet2** logging and check the event log on Windows, **syslog** on UNIX, or the operator console on OpenVMS.

   For additional troubleshooting steps for prior versions of Connectivity Series, see the Synergex KnowledgeBase article 100001607.

Connection refused (10061 or 61)

The "Connection refused" socket error, which is 10061 (WSAECONNREFUSED) on Windows and is generally 61 (ECONNREFUSED) on UNIX and OpenVMS, indicates that the SQL Connection program can't make a connection to the SQL OpenNet server. The SQL OpenNet server may not be running, it may not use the port that's specified in the connect string (or the default port if you didn't specify a port in the connect string), or the host specified in the connect string may be incorrect.

1. Use **vtxping** (or **synxfpng** with the **-x** option) to test your ability to connect to the server. (For information on **vtxping**, see "The vtxping Utility" in the "Configuring Connectivity Series" chapter of the *Installation Configuration Guide*. For information on **synxfpng**, see "The synxfpng Utility" in the "Configuring *xf*Server" chapter of the *Installation Configuration Guide*.)

   The **vtxping** and **synxfpng** utilities print reports to the screen. This information can be used by your network administrator to resolve TCP/IP network socket communication problems.

   - If you can connect, then the network, the server, and the Synergy/DE OpenNet Server service (**SynSQL**) are working. Continue with step 2.
   - If you can't connect, make sure that the server is running, that the **SynSQL** service is running on the server, and that **vtxping** or **synxfpng** is using the correct port.

2. Make sure the port number specified in the connect string matches the port number used by the SQL OpenNet server, which is either the default port number or the port number used when starting **vtxnetd** or **vtxnet2**. For information on starting **vtxnetd** or **vtxnet2**, and for information on the default port number for a server, see the "Configuring Connectivity Series" chapter of the *Installation Configuration Guide*.

UNIX

If you find that the SQL OpenNet server is running and the port is correct, it may be that server is terminating when the user that started it logs out. To run the server in the background and keep it running after you log out, use the **nohup** command. For example:

```
nohup vtxnetd &
```

For more information, see "Starting and stopping SQL OpenNet for SQL Connection" in the UNIX section of the "Configuration Connectivity Series" chapter of the *Installation Configuration Guide*.

Glossary

autocommit — A mode in which each SQL operation results in a transaction that is automatically committed after the statement is executed.

bind variable — A host variable used for sending data values to the database.

BLOB — (binary large object) Binary data that exceeds a database's normal maximum column size for binary data..

channel — A database connection control. Each connection (log-in) is maintained by channel ID.

client-side cursor — (SQL Server concept) A cursor implemented on the client for a result set that's cached on the client. The firehose cursor (though it's not a true cursor) is the only client-side cursor for SQL Server.

CLOB — (character large object) Character data that exceeds a database's normal maximum column size for character data.

commit — A procedure that finishes a transaction and makes changes permanent.

concurrency control — Methods used by relational databases to ensure that changes made to data by one user are not overwritten with changes made concurrently by other users. The two main types of concurrency control are optimistic locking and pessimistic locking.

connect string — A string use by SQL Connection to connect to and log onto a database. The connect string is passed in a call to %SSC_CONNECT. If SQL OpenNet is used for the network layer, the connect string includes a network string (see network string).

connection — A database log-in.

cursor — A processed SQL statement (one that's been parsed, optimized, etc., by the database) and/or an associated database mechanism that traverses a result set and maintains a position on a row in the result set. With SQL Connection, cursors are accessed using the number returned in the *dbcursor* argument for %SSC_OPEN.

See also logical cursor.

defined variable A host variable used for storing data values retrieved from the database.

direct connection A database connection that does not use SQL OpenNet. Direct connections must be to a local database or database client. Connect strings for direct connections start with the name or keyword of a database driver (e.g., VTX0).

drop table To delete or remove a table from a database.

dynamic cursor (SQL Server concept) A server-side cursor that reflects all changes made to the underlying data while the cursor is open. Dynamic cursors are also known as sensitive cursors, and are the only cursors you can use with SQL Server for updates and deletes. Dynamic cursors are the default cursors for VTX12_ODBC and VTX12_SQLNATIVE.

fetch An operation that retrieves a row or a set of rows from a result set. Cursor settings and type determine which row(s) will be fetched. For SQL Connection, %SSC_MOVE performs fetches.

firehose cursor (SQL Server concept) Not really a cursor, but instead the recommended mechanism when using VTX12_ODBC or VTX12_SQLNATIVE. With a firehose cursor, data is cached on the client and is forward-only and read-only. Firehose cursors minimize overhead and are faster than real cursors (dynamic, static, etc.), but they return only one row at a time, and with a firehose cursor, you can have only one active statement for a connection (though MARS may allow multiple active statements in some cases). No other statement can be executed until all results have been fetched or until the statement has been cancelled.

FOR BROWSE mode The default database locking mode. Optimistic locking is used; the lock is advisory only. The application must check whether data has changed before updating.

FOR UPDATE mode A data locking method that invokes the database engine's inherent locking method, typically a pessimistic lock. Oracle, for example, uses pessimistic locking when a SELECT FOR UPDATE operation is performed.

forward-only cursor A cursor that cannot be scrolled. With a forward-only cursor, the next row fetched will be the next row in the result set. Forward-only cursors are the only type of cursor that most databases support and are the default when using any SQL Connection database driver except VTX12_ODBC or VTX12_SQLNATIVE.

GUID
A globally unique identification number used like ROWID. GUIDs are guaranteed to be unique not only to the database, but everywhere: no other computer in the world will generate a duplicate of a GUID value.

host variable
A Synergy DBL variable used to store database data. See defined variable and bind variable.

insensitive cursor
See static cursor.

keyset-driven cursor
(SQL Server concept) A server-side cursor that reflects updates made to rows that were part of the result set when it was established. Keyset-driven cursors do not reflect deletes or inserts.

logical cursor
An SQL Connection mechanism for caching soft-closed cursors.

network string
A string that contains the information needed to connect to an SQL OpenNet service on a machine that has either the database or a client for the database. Network strings are part of connect strings (see connect string).

non-positioned cursor
A logical cursor that can't be used for positioned updates, but that is more efficient in other cases.

optimistic locking
An approach to concurrency control where the database does not lock data accessed for update or delete. Optimistic locking assumes that the front-end application will ensure data integrity. The advantage to optimistic locking is that it improves throughput. The disadvantage is that a user may be able to access rows of data only to get an error when attempting to write data to those rows because another user who concurrently accessed the data submitted changes first.

pessimistic locking
An approach to concurrency control that locks data resources for much of the duration of a transaction. For SQL Connection (when the correct commands are used), pessimistic locking locks rows from the first fetch (with %SSC_MOVE) or an insert until the transaction ends with %SSC_COMMIT, and an error is returned if another user attempts to access locked data. The advantage to pessimistic locking is that users are always able to submit changes to records and delete records without the possibility that these changes will conflict with other users' changes. The disadvantage is that throughput may suffer.

pointer
An identifier stored in a database and used by the database engine to keep track of data locations, usually by row. Analogous to a ROWID.

positioned
When discussing cursors, the current processing location for the result set. For example, if an application fetches the first two rows of a result set, the cursor will be positioned on the third row. With some databases, SQL Connection enables an application to move directly to a specific position in a cursor without performing a fetch.

positioned update
A change (update, insert, or delete) made to the underlying data at the current processing location of the result set (the row where the cursor is currently positioned). Positioned updates are invoked by a WHERE CURRENT OF clause in the update statement or by a FOR UPDATE OF clause in the select statement.

prefetch buffer
A memory resource used for improving network performance. Data from the database is retrieved and held in the prefetch buffer in anticipation of a function call.

pseudo-column
A relational database column that is part of a database table but is typically invisible to the end-user. ROWID is the most common example.

result set
A dataset that contains the results of a select statement. Result set is synonymous with rowset.

rollback
A procedure that reverses any pending changes during the current transaction, instead of committing them. Rollbacks can occur, for instance, in the event of data locking.

row locking
See optimistic locking and pessimistic locking.

row size
The aggregate size of each returned row.

ROWID
A pseudo-column used in relational database to uniquely identify each row in the database. Typically this column is not retrieved during SELECT * operations.

rowset
See result set.

scrolling cursor
A server-side cursor that enables you to determine which row will be retrieved with the next fetch. Using the %SSC_CMD option, SSQL_CMD_SCROLL, you can specify whether the next fetched row will be the current row, the first row in the result set, the last row in the result set, or a row at a given location in the result set (specified with an absolute or relative value).

sensitive cursor
See dynamic cursor.

server-side cursor (SQL Server concept) A cursor implemented on the server. Only fetched rows are sent to the client. Server-side cursors generally offer more functionality than client-side cursors, but they usually don't perform as well.

SQL OpenNet A Connectivity Series component that enables SQL Connection (and *xf*ODBC) to work in a client/server configuration. For more information, see the "Configuring Connectivity Series" chapter of the *Installation Configuration Guide*.

SQL OpenNet connection A database connection that uses SQL OpenNet for the network layer. Connect strings for SQL OpenNet connections start with "net:".

standard cursor See non-positioned cursor.

static cursor (SQL Server concept) A server-side cursor that does not reflect any changes made to the database after the result set was established. Static cursors are also known as insensitive cursors or snapshot cursors.

stored procedure A pre-compiled, ready-to-execute command stored in a database and managed as a database object. Stored procedures may limit portability.

timestamp column For SQL Server, this is analogous to the ROWID column. It uniquely identifies each row in a relational database by issuing a datetime stamp for the last moment data was committed in each row.

update To change data in a database. The terms "update" and "edit" are often used interchangeably.

vtxnetd/vtxnet2 The SQL OpenNet connection manager. For more information, see the "Configuring Connectivity Series" chapter of the *Installation Configuration Guide*.

Index

D

T