**SeeBeyond ICAN Suite**

# e*Way Intelligent Adapter for ODBC User's Guide

*Release 5.0.5 for Schema Run-time Environment (SRE)*

SEEBEYOND®

The information contained in this document is subject to change and is updated periodically to reflect changes to the applicable software. Although every effort has been made to ensure the accuracy of this document, SeeBeyond Technology Corporation (SeeBeyond) assumes no responsibility for any errors that may appear herein. The software described in this document is furnished under a License Agreement and may be used or copied only in accordance with the terms of such License Agreement. Printing, copying, or reproducing this document in any fashion is prohibited except in accordance with the License Agreement. The contents of this document are designated as being confidential and proprietary; are considered to be trade secrets of SeeBeyond; and may be used only in accordance with the License Agreement, as protected and enforceable by law. SeeBeyond assumes no responsibility for the use or reliability of its software on platforms that are not supported by SeeBeyond.

**This work is confidential and proprietary information of SeeBeyond and must be maintained in strict confidence.**

Version 20050406012757.

# Contents

Chapter 3

Chapter 4

Chapter 1

# Introduction

SeeBeyond™ developed the e*Way Intelligent Adapter for ODBC as a graphically-configurable e*Way. The ODBC e*Way implements the logic that sends Events (data) to e*Gate and queues the next Event for processing and transport to the database.

A Monk database access library is available to log into the database, issue Structured Query Language (SQL) statements, and call stored procedures. The ODBC e*Way uses Monk to execute user-supplied database access Monk scripts to retrieve information from or send information to a database. The fetched data (information) can be returned in a Monk Collaboration which simplifies the accessibility of each column in the database table. This document describes how to install and configure the ODBC e*Way.

**This Chapter Explains:**

## 1.1 Using SQL

The ODBC e*Way uses a Monk extension library to issue SQL (Structured Query Language) statements. The library contains functions to access the database and generate SQL statements. SQL is the language used to communicate with the database server to access and manipulate data. By populating a database with the data flowing through an integration engine, all the information available to an integrated delivery network (IDN) is stored for evaluation. This allows the ODBC e*Way to operate independently of the underlying DBMS (database management system).

To access the database, you execute an SQL command, which is the American National Standards Institute (ANSI) standard language for operating upon relational databases. The language contains a large set of operators for defining and manipulating tables. SQL statements can be used to create, alter, and drop tables from a database.

## 1.2 Components

The ODBC e*Way is comprised of the following:

- **stcewgenericmonk.exe**, the executable component
- Configuration files, which the e*Way Editor uses to define configuration parameters
- Monk external function scripts
- e*Way Monk functions

A complete list of installed files appears in **Table 1 on page 13** or **Table 2 on page 15**.

## 1.3 Intended Reader

The reader of this guide is presumed to be a developer or system administrator with responsibility for maintaining the e*Gate system; to have expert-level knowledge of Windows and/or UNIX operations and administration; to be thoroughly familiar with ***ODBC*** and to be thoroughly familiar with Windows-style GUI operations.

## 1.4 Supported Operating Systems

The ODBC e*Way is available on the following operating systems:

- Windows 2000, Windows XP, and Windows Server 2003
- HP-UX 11.0, and HP-UX 11i (PA-RISC), and HP-UX 11i v2.0 (11.23)
- HP Tru64 V5.1A
- IBM AIX 5.1L and 5.2
- Sun Solaris 8 and 9
- Japanese Windows 2000, Windows XP, and Windows Server 2003
- Japanese HP-UX 11.0, and HP-UX 11i (PA-RISC), and HP-UX 11i v2.0 (11.23)
- Japanese Sun Solaris 8 and 9
- Korean Windows 2000, Windows XP, and Windows Server 2003
- Korean HP-UX 11.0, and HP-UX 11i (PA-RISC), and HP-UX 11i v2.0 (11.23)
- Korean IBM AIX 5.1L and 5.2
- Korean Sun Solaris 8 and 9

# 1.5 System Requirements

To use the ODBC e*Way, you need the following:

- An e*Gate Participating Host.
- A TCP/IP network connection.

The client components of the databases with which the e*Way interfaces have their own requirements; see the appropriate documentation for more details.

## 1.5.1 External System Requirements

The ODBC e*Way supports the following external databases:

- Oracle 8.1.6
- Oracle 8.1.7
- Sybase 11
- Sybase 12
- SQL Server 7.0
- DB2

***Note:*** *DB2 support is provided by using the IBM v5 drivers for UNIX or IBM v7 drivers for Windows systems.*

Chapter 2

# Installation

This chapter describes the procedures for installing the ODBC e*Way on both Windows and UNIX systems. A list of the files and directories created by the installation are included.

**This Chapter Explains:**

## 2.1 Installation Overview

The installation procedure depends upon the operating system of the Participating Host on which you are installing this e*Way. You must have Administrator privileges to install this e*Way on either Windows or UNIX.

### 2.1.1 Installation Decisions

This section presents decisions to be made before beginning the installation. These decisions apply to both UNIX and Windows systems:

1. The operating system/platform on which the ODBC e*Way will operate.
2. The database network software required to operate the ODBC e*Way.

   For Oracle:

   - SQL *Net8

   For Sybase:

   - Open Client version11.1.x or 12
3. The Oracle networking options to be installed.

   On UNIX:

- SQL *Net8
- TCP/IP Protocol Adaptor

On Windows systems:

- SQL *Net8
- TCP/IP Adapter
- OCI

Issue the following command to determine which version of SQL *Net is installed:

On UNIX:

```
echo $ORACLE_HOME
/opt/oracle/app/oracle/product/8.1.6
```

The output shows that SQL *Plus Version 8.1.6 is installed.

### 2.1.2 Installing Client and Network Components on Windows

The following Networking Options must be installed and configured before running the ODBC e*Way:

- The Oracle8 or Oracle8i Oracle Client

**Note:** *The Oracle Client is not required for the ODBC e*Way to communicate with Microsoft SQL Server. The Oracle Client is required in order to communicate with any other database.*

- SQL*Net8 for Oracle8 and 8i
- TCP/IP Protocol Adapter
- OCI (Oracle Call Interface)
- The Sybase Open Client

**Note:** *The ODBC e*Way requires a 32-bit version of the Oracle Client. The 64-bit Oracle Client is not compatible with this e*Way.*

## 2.2 Installing the ODBC e*Way on Windows Systems

### 2.2.1 Pre-installation

1. Exit all Windows programs before running the setup program, including any anti-virus applications.
2. You must have Administrator privileges to install this e*Way.

## 2.2.2 Installation Procedure

### To install the ODBC e*Way on a Windows systems

1. Log in as an Administrator on the workstation on which you want to install the e*Way.
2. Insert the e*Way installation CD-ROM into the CD-ROM drive.
3. If the CD-ROM drive's "Autorun" feature is enabled, the setup application should launch automatically; skip ahead to step 4. Otherwise, use the Windows Explorer or the Control Panel's **Add/Remove Applications** feature to launch the file **setup.exe** on the CD-ROM drive.
4. The InstallShield setup application will launch. Follow the on-screen instructions to install the e*Way.

**Note:** *Be sure to install the e*Way files in the suggested "client" installation directory. The installation utility detects and suggests the appropriate installation directory.* ***Unless you are directed to do so by SeeBeyond support personnel, do not change the suggested "installation directory" setting.***

The ODBC e*Way CD-ROM contains the following files, which the InstallShield Wizard copies to the indicated directories on your computer, creating them if necessary.

These files are installed in the Registry during your initial installation. The first time you access the e*Way to configure it, the following files (with the exception of all the Monk files) move to the Client directory.

**Table 1** Installation Directories and Files (Windows)

| Install Directory | Files |
|---|---|
| bin\ | stcewgenericmonk.exe<br>stcstruct.exe<br>stc_dbapps.dll<br>stc_dbmonkext.dll<br>stc_dbodbc.dll<br>stccdbctest.exe |
| configs\stcewgenericmonk\ | dart.def<br>dartRule.txt |
| monk_library\ | dart.gui |

**Table 1** Installation Directories and Files (Windows)

| Install Directory | Files |
|---|---|
| monk_library\dart\ | db-struct-bulk-insert.monk<br>db-struct-call.monk<br>db-struct-execute.monk<br>db-struct-fetch.monk<br>db-struct-insert.monk<br>db-struct-select.monk<br>db-struct-update.monk<br>db-stdver-eway-funcs.monk<br>odbcmsg.ssc<br>odbcmsg-display.monk<br>db_bind.monk<br>db-sanitize-symbol.monk |

# 2.3 Installing the ODBC e*Way on UNIX

## 2.3.1 Pre-installation

You do not require root privileges to install this e*Way. Log in under the user name that you wish to own the e*Way files. Be sure that this user has sufficient privilege to create files in the e*Gate directory tree.

## 2.3.2 Installation Procedure

**To install the ODBC e*Way on a UNIX system:**

1. Log in on the workstation containing the CD-ROM drive, and insert the CD-ROM into the drive.
2. If necessary, mount the CD-ROM drive.
3. At the shell prompt, type

   **cd /cdrom**
4. Start the installation script by typing:

   **setup.sh**
5. A menu of options will appear. Select the **e*Gate Add-on Application** option. Then, follow any additional on-screen directions.

**Note:** *Be sure to install the e*Way files in the suggested "client" installation directory. The installation utility detects and suggests the appropriate installation directory.* ***Unless you are directed to do so by SeeBeyond support personnel, do not change the suggested "installation directory" setting.***

The CD-ROM contains the following files, which are copied to the indicated path on your computer. Refer to the installation instructions for e*Gate for the most up-to-date information.

**Table 2** Installation Directories and Files (UNIX)

| Install Directory | Files |
|---|---|
| bin/ | stcewgenericmonk.exe<br>stc_dbapps.dll<br>stc_dbmonkext.dll<br>stc_dbodbc.dll<br>stc_dbctest.exe<br>stcstruct.exe<br>stc_dbodbc.dll |
| configs/stcewgenericmonk/ | dart.def<br>dartRule.txt |
| monk_library | dart.gui |
| monk_library/dart/ | db-struct-bulk-insert.monk<br>db-struct-call.monk<br>db-struct-execute.monk<br>db-struct-fetch.monk<br>db-struct-insert.monk<br>db-struct-select.monk<br>db-struct-update.monk<br>db-stdver-eway-funcs.monk<br>db_bind.monk<br>odbcmsg.ssc<br>odbcmsg-display.monk<br>db-sanitize-symbol.monk |

# 2.4 Merant 4.0 ODBC Drivers

This section covers installation, setup, and testing of the Merant ODBC Drivers. If you are using either the Japanese or Korean version of the ODBC e*Way, you will need to install the Merant 3.6 ODBC drivers. If you are using DB2 with the ODBC e*Way, you will need to install the Merant 3.6 ODBC drivers.

## 2.4.1 Setting up the Shared Library Search Path

You must set up the shared library search path used by both the ODBC e*Way and the Sybase Open Client. The library search path environment variables are required to be set so that the ODBC core components and drivers can be located at the time of execution.

You can define these environment variables in *.cshrc* in the C shell or *.profile* in the Korn/Bash shell. Follow these scripts when setting up the shared library search path:

### Korn/Bash Shell

```
if [ "$LD_LIBRARY_PATH" = "" ]; then
  LD_LIBRARY_PATH=/opt/odbc/lib
else
  LD_LIBRARY_PATH=/opt/odbc/lib:$LD_LIBRARY_PATH
fi
export LD_LIBRARY_PATH
```

### C Shell

```
if ($?LD_LIBRARY_PATH == 1) then
  setenv LD_LIBRARY_PATH  /opt/odbc/lib:${LD_LIBRARY_PATH}
else
  setenv LD_LIBRARY_PATH  /opt/odbc/lib
endif
```

## 2.4.2 Setting up the ODBC Data Source Definition File

In the UNIX environment, there is no ODBC Administrator. To configure a data source, you must edit the odbc.ini file, a plain text file that is normally located in the user's $HOME directory. This file is maintained using any text editor to define data source entries as described in the "Connecting to a Data Source Using a Connection String" section of each driver's chapter. A sample file (odbc.ini) is located in the driver installation directory.

UNIX support of the database drivers also allows the use of a centralized **.odbc.ini** file that a system administrator can control. This is accomplished by setting the environment variable ODBCINI to point to the fully qualified pathname of the centralized file.

The search order for the location of the **.odbc.ini** file is as follows:

1 Check $ODBCINI

2 Check $HOME/.odbc.ini

There must be an ODBC section in the **.odbc.ini** file that includes the InstallDir keyword. The value of this keyword must be the path to the directory under which the /lib and /messages directories are contained. For example, if you choose the default install directory, then the following line must be in the [ODBC] section:

```
InstallDir=/opt/odbc
```

### Sample .odbc.ini File

The following is an **.odbc.ini** file which contains some sample values to use when setting these environment variables.

```
[ODBC Data Sources]
DB2 Wire Protocol=DataDirect 4.00 DB2 Wire Protocol Driver
dBase=DataDirect 4.0 dBaseFile(*.dbf)
Informix=DataDirect 4.0 Informix
Informix Wire Protocol=DataDirect 4.0 Informix Wire Protocol
Oracle=DataDirect 4.0 Oracle
Oracle Wire Protocol=DataDirect 4.0 Oracle Wire Protocol
SQLServer Wire Protocol=DataDirect 4.0 SQL Server Wire Protocol
Sybase Wire Protocol=DataDirect 4.0 Sybase Wire Protocol
Text=DataDirect 4.0 TextFile(*.*)
```

```
[DB2 Wire Protocol]
Driver=/opt/odbc/lib/DGdb217.so
Description=DB2 Wire Protocol Driver
LogonID=uid
Password=pwd
DB2AppCodePage=1252
ServerCharSet=1252
IpAddress=db2host
Database=db
TcpPort=50000
Package=db2package
Action=REPLACE
QueryBlockSize=8
CharSubTypeType=SYSTEM_DEFAULT
ConversationType=SINGLE_BYTE
CloseConversation=DEALLOC
UserBufferSize=32
MaximumClients=35
GrantExecute=1
GrantAuthid=PUBLIC
OEMANSI=1
DecimalDelimiter=PERIOD
DecimalPrecision=15
StringDelimiter=SINGLE_QUOTE
IsolationLevel=CURSOR_STABILITY
ResourceRelease=DEALLOCATION
DynamicSections=32
Trace=0
WithHold=0

[dBase]
Driver=/opt/odbc/lib/DGdbf17.so
Description=dBaseFile(*.dbf)
Database=/opt/odbc/demo
CacheSize=4
Locking=RECORD
CreateType=dBASE5
IntlSort=0
UseLongNames=1
UseLongQualifiers=1
ApplicationUsingThreads=1

[Informix]
Driver=/opt/odbc/lib/DGinf17.so
Description=Informix
Database=db
LogonID=uid
Password=pwd
ServerName=informixserver
HostName=informixhost
Service=online
Protocol=onsoctcp
EnableInsertCursors=0
GetDBListFromInformix=0
CursorBehavior=0
CancelDetectInterval=0
TrimBlankFromIndexName=1
ApplicationUsingThreads=1

[Informix Wire Protocol]
Driver=/opt/odbc/lib/DGifcl17.so
Description=Informix Wire Protocol
Database=db
```

```
LogonID=uid
Password=pwd
HostName=informixhost
PortNumber=1500
ServerName=informixserver
EnableInsertCursors=0
GetDBListFromInformix=0
CursorBehavior=0
CancelDetectInterval=0
TrimBlankFromIndexName=1
ApplicationUsingThreads=1

[Oracle]
Driver=/opt/odbc/lib/DGor817.so
Description=Oracle
LogonID=uid
Password=pwd
ServerName=oraclehost
CatalogOptions=0
ProcedureRetResults=0
EnableDescribeParam=0
EnableStaticCursorsForLongData=0
ApplicationUsingThreads=1

[Oracle Wire Protocol]
Driver=/opt/odbc/lib/DGora17.so
Description=Oracle Wire Protocol
LogonID=uid
Password=pwd
HostName=oracleserver
PortNumber=1521
SID=oraclesid
CatalogOptions=0
ProcedureRetResults=0
EnableDescribeParam=0
EnableStaticCursorsForLongData=0
ApplicationUsingThreads=1

[SQLServer Wire Protocol]
Driver=/opt/odbc/lib/DGmsss17.so
Description=SQL Server Wire Protocol
Database=db
LogonID=uid
Password=pwd
Address=sqlserverhost,1433
QuotedId=No
AnsiNPW=No

[Sybase Wire Protocol]
Driver=/opt/odbc/lib/DGase17.so
Description=Sybase Wire Protocol
Database=db
LogonID=uid
Password=pwd
NetworkAddress=serverhost,4100
EnableDescribeParam=1
EnableQuotedIdentifiers=0
OptimizePrepare=1
RaiseErrorPositionBehavior=0
SelectMethod=0
ApplicationUsingThreads=1

[Text]
Driver=/opt/odbc/lib/DGtxt17.so
```

```
Description=TextFile(*.*)
Database=/opt/odbc/demo
AllowUpdateAndDelete=0
CacheSize=4
CenturyBoundary=20
FileOpenCache=0
UndefinedTable=GUESS
IntlSort=0
ScanRows=25
TableType=Comma
UseLongQualifiers=0
ApplicationUsingThreads=1

[ODBC]
Trace=0
TraceFile=odbctrace.out
TraceDll=/opt/odbc/odbctrac.so
InstallDir=/opt/odbc
ConversionTableLocation=/opt/odbc/tables
UseCursorLib=0
```

***Caution:*** *The "Trace" value must be set to* ***0****. Setting this value to* ***1*** *can cause some third-party applications to interfere with e*Gate.*

### Optional Environment Variables

Many of the drivers must have environment variables set as required by the database client components used by the drivers. Consult the system requirements in each of the individual driver sections for additional information pertaining to individual driver requirements.

## 2.4.3 The ivtestlib Tool

The ivtestlib tool is provided to help diagnose configuration problems (such as environment variables not correctly set or missing database management system client components) in the UNIX environment. This command will attempt to load a specified ODBC driver and will print out all available error information if the load fails. For example, the following command will attempt to load the Oracle driver on Solaris.

```
ivtestlib /opt/odbc/lib/dgor816.so
```

The executable [ivtestlib] is located in the `/opt/odbc/bin` directory

## 2.4.4 Testing the ODBC Driver

To test if the driver is running correctly, log in as the client (e.g., ODBC) and run the test program **stcodbctest.exe**.

```
stcodbctest data_source user_name password
```

#### Example stcodbctest ODBC e*Way

Here's a typical output message for **stcodbctest**:

```
Environment handle allocated.
Connection handle allocated.
Data source: SQL4.0 found.
Database connection established.
```

```
ODBC Driver Information:
        Driver Name                     : DGSS617.DLL
        Driver Version                  : 04.00.0004
        Driver Manager ODBC Version     : 03.52.0000
        Driver ODBC Version             : 03.51
        Driver ODBC API Conformance     : Level 1 supported
        Driver ODBC SQL Conformance     : Core grammar supported
        Driver ODBC Procedure Support   : Yes

DBMS Product Information:
        DBMS Name                       : Microsoft SQL Server
        DBMS Version                    : 08.00.0194

Data Source Information:
        Data Source Name                : SQL4.0
        Server Name                     : anu2000
        Database Name                   : pubs
        User Name                       : dbo
        Transaction Support             : Both DML and DDL statements
are supported

ODBC Function Information:
        SQLNumResultCols                : supported
        SQLDescribeCol                  : supported
        SQLBindCol                      : supported
        SQLNumParams                    : supported
        SQLDescribeParam                : not supported
        SQLBindParameter                : supported
        SQLProcedures                   : supported
        SQLProcedureColumns             : supported

Database connection terminated.
Connection handle freed.
```

It is important that all of the above ODBC Function Information parameters indicate that they are supported.

**Note:** *In order to assure that the latest statement functionality is available, the SQLDescribeParam line item must be present, and indicate "supported".*

## 2.5 Installing the ODBC Drivers for Compaq

To operate the ODBC e*Way on Compaq Tru64 systems, obtain the Compaq ODBC drivers from the following location:

**http://tru64unix.compaq.com/data-access/download.htm**

You will be required to register prior to downloading the drivers.

## 2.6 Oracle Network Components

Install the following Oracle networking options when running as a client database.

SQL *Net8 (Oracle8)

TCP/IP Protocol Adapter

**Note:** *Install SQL *PLUS to test out the connection.*

## 2.6.1 SQL *Net V2 Configuration Files

Before you can configure SQL *Net8 you must have the following files ready:

- listener.ora
- tnsnames.ora
- sqlnet.ora

### Example Listener configuration file—listener.ora

```
# LISTENER.ORA Configuration
File:/opt/oracle/app/oracle/product/8.1.6/network/admin/listener.ora
# Generated by Oracle configuration tools.

LISTENER =
  (DESCRIPTION_LIST =
    (DESCRIPTION =
      (ADDRESS_LIST =
        (ADDRESS = (PROTOCOL = IPC)(KEY = EXTPROC))
      )
      (ADDRESS_LIST =
        (ADDRESS = (PROTOCOL = TCP)(HOST = circe)(PORT = 1521))
      )
    )
    (DESCRIPTION =
      (PROTOCOL_STACK =
        (PRESENTATION = GIOP)
        (SESSION = RAW)
      )
      (ADDRESS = (PROTOCOL = TCP)(HOST = circe)(PORT = 2481))
    )
  )

SID_LIST_LISTENER =
  (SID_LIST =
    (SID_DESC =
      (SID_NAME = PLSExtProc)
      (ORACLE_HOME = /opt/oracle/app/oracle/product/8.1.6)
      (PROGRAM = extproc)
    )
    (SID_DESC =
      (ORACLE_HOME = /opt/oracle/app/oracle/product/8.1.6)
      (SID_NAME = orcl816)
    )
  )
```

LISTENER is the default listener name, which is recommended by Oracle in a standard installation that requires only one listener on a machine.

Listener address section ADDRESS specifies what address to listen to. The listener listens for inter-process calls (IPC's) as well as calls from other nodes.

Two IPC addresses are created for each database that a listener listens to. In one, the key value is equal to the service name (e.g., finance.world). It is used for connections from other applications on the same node. The other IPC address (e.g., orcl) is used by the database dispatcher to identify the listener.

For communication with other nodes, listener listens to the host (e.g., finance.company.com) at a particular port (e.g., 1521) using the specified protocol (e.g., TCP/IP).

The section SID_LIST_LISTENER is used to describe the SID (system identified) of the databases (e.g., orcl) on which the listener listens. The service name (e.g., finance.world) is used as the global name.

The control parameter STARTUP_WAIT_TIME_LISTENER sets the number of seconds that the listener sleeps before responding to the first listener control status command. This feature assures that a listener with a slow protocol will have had time to start up before responding to a status request. The default is 0.

CONNECT_TIMEOUT_LISTENER sets the number of seconds that the listener waits to get a valid SQL*Net connection request before dropping the connection.

TRACE_LEVEL_LISTENER indicates the level of detail the trace facility records for listener events. ADMIN is the highest.

### Example Client file—tnsnames.ora

```
# TNSNAMES.ORA Configuration
File:/opt/oracle/app/oracle/product/8.1.6/network/admin/tnsnames.ora
# Generated by Oracle configuration tools.

CIRCE =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)(HOST = circe)(PORT = 1521))
    (CONNECT_DATA = (SERVICE_NAME = orcl816))
  )
ENIGMA =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)(HOST = enigma)(PORT = 1521))
    (CONNECT_DATA = (SID = orcl8))
  )
LAMBDA =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)(Host = lambda)(Port = 1521))
    (CONNECT_DATA = (SERVICE_NAME = LAMBDA))
  )
```

All connect distributors are assigned service names (e.g., finance.world). The user specifies the service name to identify the service to which the user wants to connect. The ADDRESS section specifies the listener address. See listener.ora above for listener address.

The CONNECT_DATA section specifies the SID (system identified) by the remote database.

### Network component file—sqlnet.ora

```
File: sqlnet.ora
################
# Filename......: sqlnet.ora
# Node..........: local.world
# Date..........: 24-MAR-98 13:21:20
################
AUTOMATIC_IPC = OFF
TRACE_LEVEL_CLIENT = OFF
```

```
names.directory_path = (TNSNAMES)
names.default_domain = world
name.default_zone = world
sqlnet.expire_time = 10
```

The sqlnet.expire_time parameter determines how often SQL*Net sends a probe to verify that a client-server connection is still active. A value of 10 (minutes) is recommended by Oracle.

After you have generated the required configuration files, do the following:

- On the server side, move all three files to:

  ```
  $ORACLE_HOME/network/admin
  ```

- On the client side, distribute tnsnames.ora and sqlnet.ora and put them in:

  ```
  $ORACLE_HOME/network/admin
  ```

Verify that the file/etc/services has the entry LISTENER 1521/tcp.

## 2.6.2 Testing the SQL *Net Configuration

Before you can use SQL*Net with the server, you need to start a listener on the server. A listener is used by SQL*Net on the server side to receive an incoming connection from SQL*Net clients.

To start a listener, enter the following command in the server:

```
lsnrctl start listener name
```

Where <listener name> is optional for the default listener. For example, to start up the default LISTENER in the machine enterprise, the command would be:

```
lsnrct start
```

When you are running as a client, if the listener starts up successfully, you can use SQL*Plus on the client side to test whether SQL*Net is configured properly by establishing a connection with the server. For example:

```
sqlplus hcaufield/phoebie@oracle.world
```

This command will start up sqlplus in the client machine enterprise and connect to the server specified by oracle.world as user *hcaufield* with password *phoebie*. The syntax of the command is:

```
sqlplus user name/password@service name
```

**Note:** *The $ORACLE_HOME/network/admin/tnsnames.ora defines the service name for each Oracle data source.*

## 2.6.3 Troubleshooting Checklist

- Ensure you have protocol-level connectivity (for TCP/IP, connectivity can be tested using the ping utility).
- Ensure client machine has configuration files (TNSNAMES.ORA and SQLNET.ORA) in the $ORACLE_HOME/network/admin directory. Also check that the server has the configuration files (LISTENER.ORA, TNSNAMES.ORA, and SQLNET.ORA) in its default directory.

- Check whether the listener is "listening" for the same protocol the client is trying to connect through.
- Verify that both server and client are running either Net8 or SQL *Net V2. Net 8 and SQL *Net V2 can communicate to each other. Verify the version by running the Oracle Universal Installer.
- Ensure that you have the necessary Net8 protocol support installed. Verify by running the Oracle Universal Installer.
- Verify that the net service name is typed correctly. The net service name should be listed under the Net Service Names folder in the Net8 Assistant.
- check to see if the default domain in your profile is set. If it is, then the net service names will have the same value appended to them. For example, if the default domain in your profile is set to ACME.COM, then all net service names will have the.ACME.COM extension appended.
- If you are using TCP/IP, try replacing the HOST name in the net service name address with the IP address of the server machine.

For more information on specific error messages or technical bulletins on errors received when performing these diagnostics tests, refer:

- The Net8 Administrator’s Guide

Chapter 3

# Configuration

Before you can run the ODBC e*Way, you must configure it using the e*Way Editor, which is accessed from the e*Gate Schema Designer GUI. The ODBC e*Way package includes a default configuration file which you can modify using this window.

This chapter describes the procedure for configuring a new e*Way. You can also edit an existing e*Way and rename an e*Way. Procedures for creating and editing e*Gate components are provided in the Schema Designer's online help.

**This Chapter Explains:**

## 3.1 Configuration Overview

Before you can run the ODBC e*Way, you must configure it using the e*Way Edit Settings window, which is accessed form the e*Gate Schema Designer GUI. The ODBC e*Way package includes a default configuration file which you can modify using this window.

## 3.2 e*Way Configuration Parameters

e*Way configuration parameters are set using the e*Way Editor.

**To change e*Way configuration parameters:**

1. In the Schema Designer's Component editor, select the e*Way you want to configure and display its properties.
2. Under **Configuration File**, click **New** to create a new file, **Find** to select an existing configuration file, or **Edit** to edit the currently selected file.

3. In the **Additional Command Line Arguments** box, type any additional command line arguments that the e*Way may require, taking care to insert them *at the end* of the existing command-line string. Be careful not to change any of the default arguments unless you have a specific need to do so.

For more information about how to use the e*Way Editor, see the e*Way Editor's online Help or the *Working with e*Ways* chapter in the *e*Gate Integrator User's Guide*.

The e*Way's configuration parameters are organized into the following sections:

- General Settings
- Communication Setup
- Monk Configuration
- Database Setup

## 3.2.1 General Settings

The General Settings control basic operational parameters.

### Journal File Name

#### Description

Specifies the name of the journal file, which will store messages that are not picked up from the queue.

#### Required Values

A valid filename, optionally including an absolute path (for example, **c:\temp\filename.txt**). If an absolute path is not specified, the file will be stored in the e*Gate "SystemData" directory. See the *e*Gate Integrator System Administration and Operations* Guide for more information about file locations. If the directory does not exist, the e*Way will create it.

#### Additional Information

The Journal File is used for the following conditions:

- Journal a message when it exceeds the number of retries.
- When its receipt is due to an external error, but Forward External Errors is set to **No**. (See **"Forward External Errors" on page 27** for more information.)

### Max Resends Per Message

#### Description

Specifies the maximum number of times the e*Way will attempt to resend a message to the external system after receiving an error. When this maximum number is reached, the message is considered "failed" and will be written to the journal file.

#### Required Values

An integer between 1 and 1,024. The default is 5.

### Max Failed Messages

#### Description

Specifies the maximum number of failed messages the e*Way will allow. When the specified number of failed messages is reached and journaled, the e*Way will shut down and exit.

#### Required Values

An integer between 1 and 1,024. The default is 3.

### Forward External Errors

#### Description

Selects whether error messages that begin with the string **DATAERR** that are received from the external system will be queued to the e*Way's configured queue. See **"Exchange Data with External Function" on page 41** for more information.

#### Required Values

**Yes** or **No**. The default value, **No**, specifies that error messages will not be forwarded.

See **"Schedule-driven data exchange functions" on page 36** for information about how the e*Way uses this function.

## 3.2.2 Communication Setup

The Communication Setup parameters control the schedule by which the e*Way obtains data from the external system.

***Note:*** *The schedule you set using the e*Way's properties in the Schema Designer controls when the e*Way executable will run. The schedule you set within the parameters discussed in this section (using the e*Way Editor) determines when data will be exchanged. Be sure you set the "exchange data" schedule to fall within the "run the executable" schedule.*

### Start Exchange Data Schedule

#### Description

Establishes the schedule to invoke the e*Way's **Exchange Data with External** function.

#### Required Values

One of the following:

- One or more specific dates/times
- A single repeating interval (such as yearly, weekly, monthly, daily, or every *n* seconds).

.Since months do not all contain equal numbers of days, be sure not to provide boundaries that would cause an invalid date selection (i.e. the 30th of every month would not include February).

**Also required:** If you set a schedule using this parameter, you must also define all three of the following:

- **Exchange Data with External Function** on page 41
- **Positive Acknowledgment Function** on page 43
- **Negative Acknowledgment Function** on page 44

If you do not do so, the e*Way will terminate execution when the schedule attempts to start.

### Additional Information

When the schedule starts, the e*Way determines whether it is waiting to send an ACK or NAK to the external system (using the Positive and Negative Acknowledgment functions) and whether the connection to the external system is active. If no ACK/NAK is pending and the connection is active, the e*Way immediately executes the **Exchange Data with External** function. Thereafter, the **Exchange Data with External** function will be called according to the **Exchange Data Interval** parameter until the **Stop Exchange Data Schedule** time is reached.

See **"Exchange Data with External Function" on page 41**, **"Exchange Data Interval" on page 28**, and **"Stop Exchange Data Schedule" on page 28** for more information.

## Stop Exchange Data Schedule

### Description

Establishes the schedule to stop data exchange.

### Required Values

One of the following:

- One or more specific dates/times
- A single repeating interval (such as yearly, weekly, monthly, daily, or every *n* seconds).

Since months do not all contain equal numbers of days, be sure not to provide boundaries that would cause an invalid date selection (i.e. the 30th of every month would not include February).

## Exchange Data Interval

### Description

Specifies the number of seconds the e*Way waits between calls to the **Exchange Data with External** function during scheduled data exchanges.

### Required Values

An integer between 0 and 86,400. The default is 120.

### Additional Information

If **Zero Wait Between Successful Exchanges** is set to **Yes** and the **Exchange Data with External Function** returns data, The **Exchange Data Interval** setting will be ignored and the e*Way will invoke the **Exchange Data with External Function** immediately.

If this parameter is set to zero, there will be no exchange data schedule set and the **Exchange Data with External Function** will never be called.

See **"Start Exchange Data Schedule" on page 27** and **"Stop Exchange Data Schedule" on page 28** for more information about the data-exchange schedule.

## Down Timeout

### Description

Specifies the number of seconds that the e*Way will wait between calls to the **External Connection Establishment function**. See **"External Connection Establishment Function" on page 42** for more information.

### Required Values

An integer between 1 and 86,400. The default is 15.

## Up Timeout

### Description

Specifies the number of seconds that the e*Way will wait between calls to the **External Connection Verification function** to verify that the connection is still up. See **"External Connection Verification Function" on page 42** for more information.

### Required Values

An integer between 1 and 86,400. The default is 15.

## Resend Timeout

### Description

Specifies the number of seconds the e*Way will wait between attempts to resend a message to the external system, after receiving an error message from the external.

### Required Values

An integer between 1 and 86,400. The default is 10.

## Zero Wait Between Successful Exchanges

### Description

Selects whether to initiate data exchange after the **Exchange Data Interval** or immediately after a successful previous exchange.

### Required Values

**Yes** or **No**. If this parameter is set to **Yes**, the e*Way will immediately invoke the **Exchange Data with External function** if the previous exchange function returned data. If this parameter is set to **No**, the e*Way will always wait the number of seconds specified by **Exchange Data Interval** between invocations of the **Exchange Data with External function**. The default is **No**.

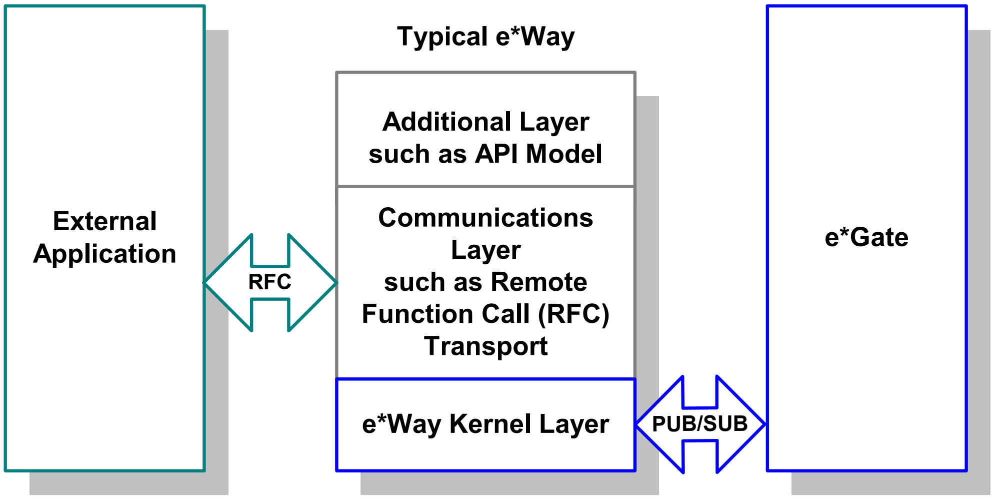
See **“Exchange Data with External Function” on page 41** for more information.

## 3.2.3 Monk Configuration

The parameters in this section help you set up the information required by the e*Way to utilize Monk for communication with the external system.

Architecturally, an e*Way can be viewed as a multi-layered structure, consisting of one or more layers that handle communication with the external application, built upon an e*Way Kernel layer that manages the processing of data and subscribing or publishing to other e*Gate components (see Figure 1).

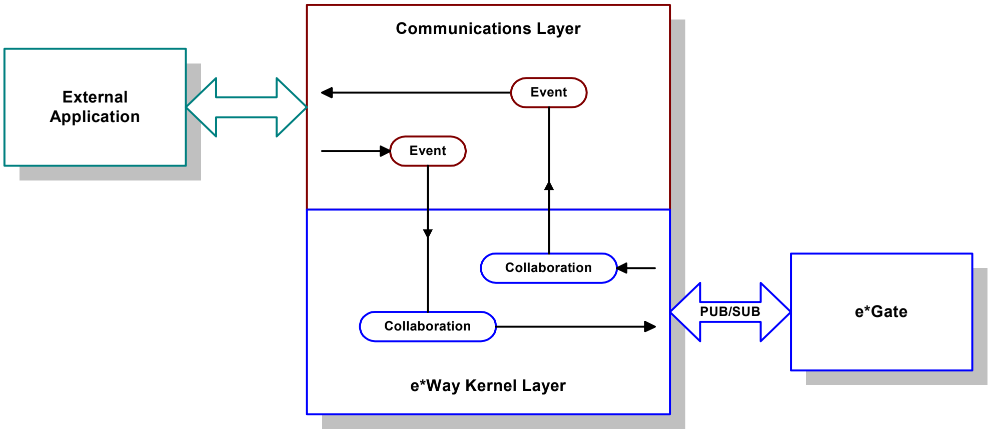
**Figure 1** Typical e*Way Architecture


Typical e*Way
External Application
RFC
Additional Layer such as API Model
Communications Layer such as Remote Function Call (RFC) Transport
e*Way Kernel Layer
PUB/SUB
e*Gate

Each layer contains Monk scripts and/or functions, and makes use of lower-level Monk functions residing in the layer beneath. You, as user, primarily use the highest-level functions, which reside in the upper layer(s).

The upper layers of the e*Way use Monk functions to start and stop scheduled operations, exchange data with the external system, package data as e*Gate “Events,” send those Events to Collaborations, and manage the connection between the e*Way and the external system (see **Figure 2 on page 31**).

**Figure 2** Basic e*Way Operations



Configuration options that control the Monk environment and define the Monk functions used to perform these basic e*Way operations are discussed in **Chapter 4**. You can create and modify these functions using the SeeBeyond Collaboration Rules Editor or a text editor (such as Microsoft Wordpad or Notepad).

The upper layers of the e*Way are single-threaded. Functions run serially, and only one function can be executed at a time. The e*Way Kernel is multi-threaded, with one executable thread for each Collaboration. Each thread maintains its own Monk environment; therefore, information such as variables, functions, path information, and so on cannot be shared between threads.

The basic set of e*Way Kernel Monk functions is described in **Chapter 5**. Generally, e*Way Kernel Monk functions should be called directly only when there is a specific need not addressed by higher-level Monk functions, and should be used only by experienced developers.

## Basic e*Way Processes

The Monk functions in the “communications half” of the e*Way fall into the following groups:

| Type of Operation | Name |
|---|---|
| Initialization | **Startup Function** on page 40<br>(also see **Monk Environment Initialization File** on page 39) |
| Connection | **External Connection Establishment Function** on page 42<br>**External Connection Verification Function** on page 42<br>**External Connection Shutdown Function** on page 43 |

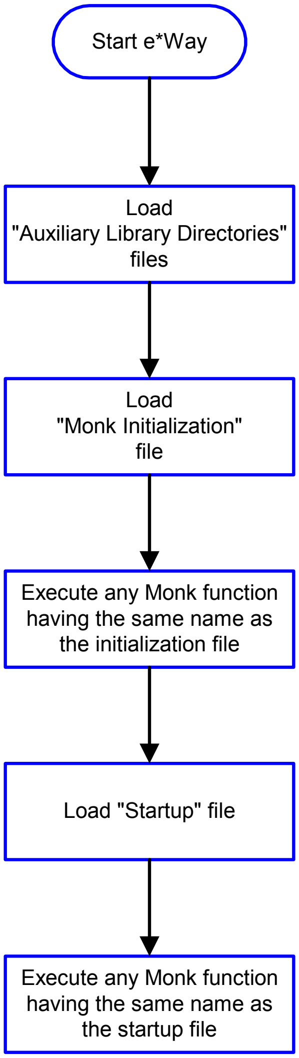
| Type of Operation | Name |
|---|---|
| Schedule-driven data exchange | **Exchange Data with External Function** on page 41<br>**Positive Acknowledgment Function** on page 43<br>**Negative Acknowledgment Function** on page 44 |
| Shutdown | **Shutdown Command Notification Function** on page 45 |
| Event-driven data exchange | **Process Outgoing Message Function** on page 40 |

A series of figures on the next several pages illustrates the interaction and operation of these functions.

### Initialization Functions

Figure 3 illustrates how the e*Way executes its initialization functions.

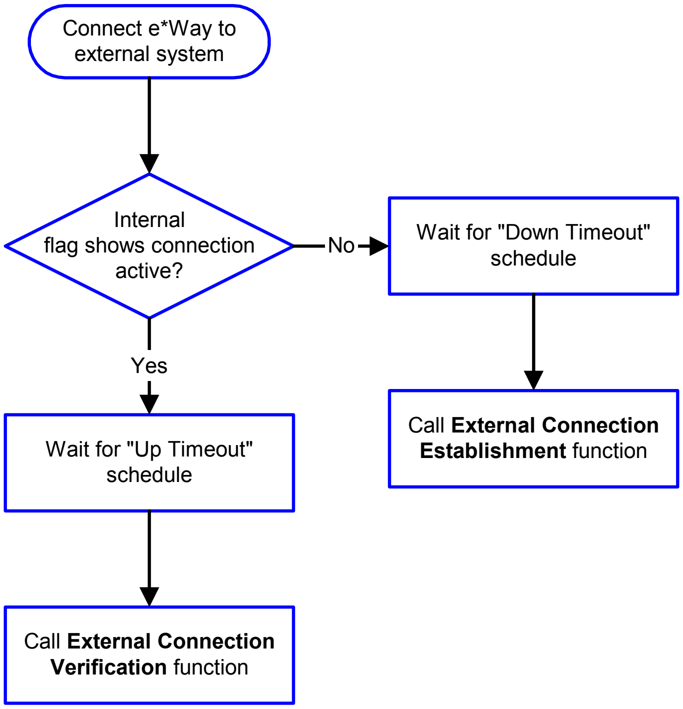
**Figure 3** Initialization Functions


Start e*Way
Load "Auxiliary Library Directories" files
Load "Monk Initialization" file
Execute any Monk function having the same name as the initialization file
Load "Startup" file
Execute any Monk function having the same name as the startup file

### Connection Functions

Figure 4 illustrates how the e*Way executes the connection establishment and verification functions.

**Figure 4** Connection establishment and verification functions

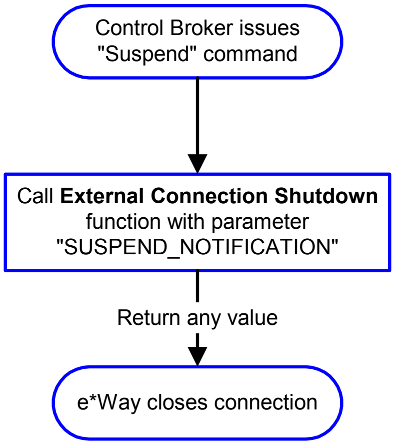

Connect e*Way to external system
Internal flag shows connection active?
No
Yes
Wait for "Down Timeout" schedule
Call **External Connection Establishment** function
Wait for "Up Timeout" schedule
Call **External Connection Verification** function

**Note:** *The e*Way selects the connection function based on an internal "up/down" flag rather than a poll to the external system. See* **Figure 6 on page 36** *and* **Figure 8 on page 38** *for examples of how different functions use this flag.*

*User functions can manually set this flag using Monk functions. See and* **send-external-up** *on page 85* **send-external-down** *on page 84 for more information.*

Figure 5 illustrates how the e*Way executes its "connection shutdown" function.

**Figure 5** Connection shutdown function


Control Broker issues "Suspend" command
Call **External Connection Shutdown** function with parameter "SUSPEND_NOTIFICATION"
Return any value
e*Way closes connection

### Schedule-driven Data Exchange Functions

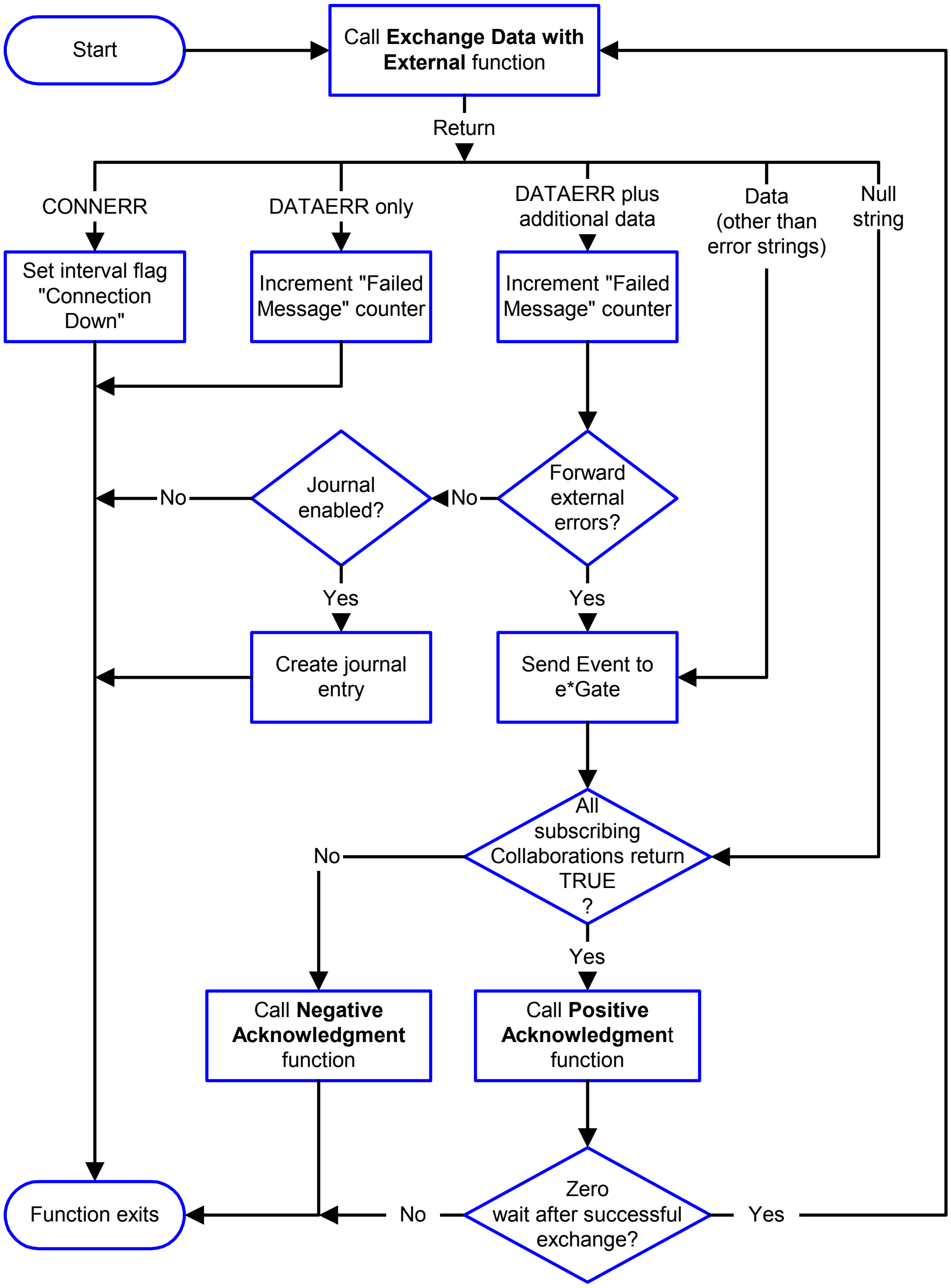
Figure 6 (on the next page) illustrates how the e*Way performs schedule-driven data exchange using the **Exchange Data with External Function**. The **Positive Acknowledgement Function** and **Negative Acknowledgement Function** are also called during this process.

“Start” can occur in any of the following ways:

- The “Start Data Exchange” time occurs
- Periodically during data-exchange schedule (after “Start Data Exchange” time, but before “Stop Data Exchange” time), as set by the Exchange Data Interval
- The **start-schedule** Monk function is called

After the function exits, the e*Way waits for the next “start schedule” time or command.

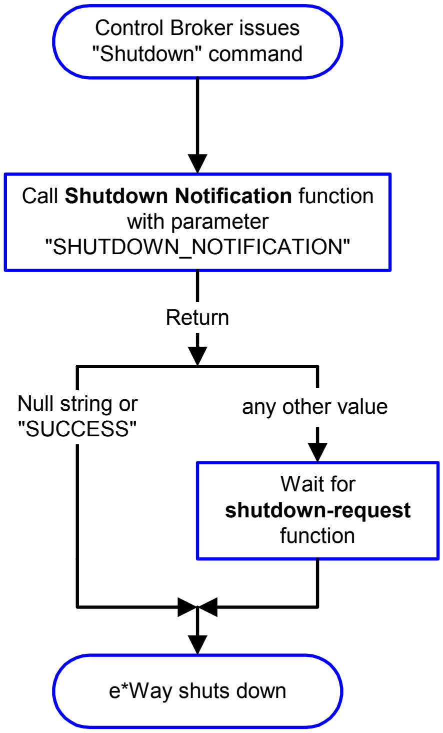
**Figure 6** Schedule-driven data exchange functions


Start
Call **Exchange Data with External** function
Return
CONNERR
DATAERR only
DATAERR plus additional data
Data (other than error strings)
Null string
Set interval flag "Connection Down"
Increment "Failed Message" counter
Increment "Failed Message" counter
Journal enabled?
No
No
Forward external errors?
Yes
Yes
Create journal entry
Send Event to e*Gate
All subscribing Collaborations return TRUE ?
No
Yes
Call **Negative Acknowledgment** function
Call **Positive Acknowledgment** function
Function exits
No
Zero wait after successful exchange?
Yes

## Shutdown Functions

Figure 7 illustrates how the e*Way implements the shutdown request function.

**Figure 7** Shutdown functions

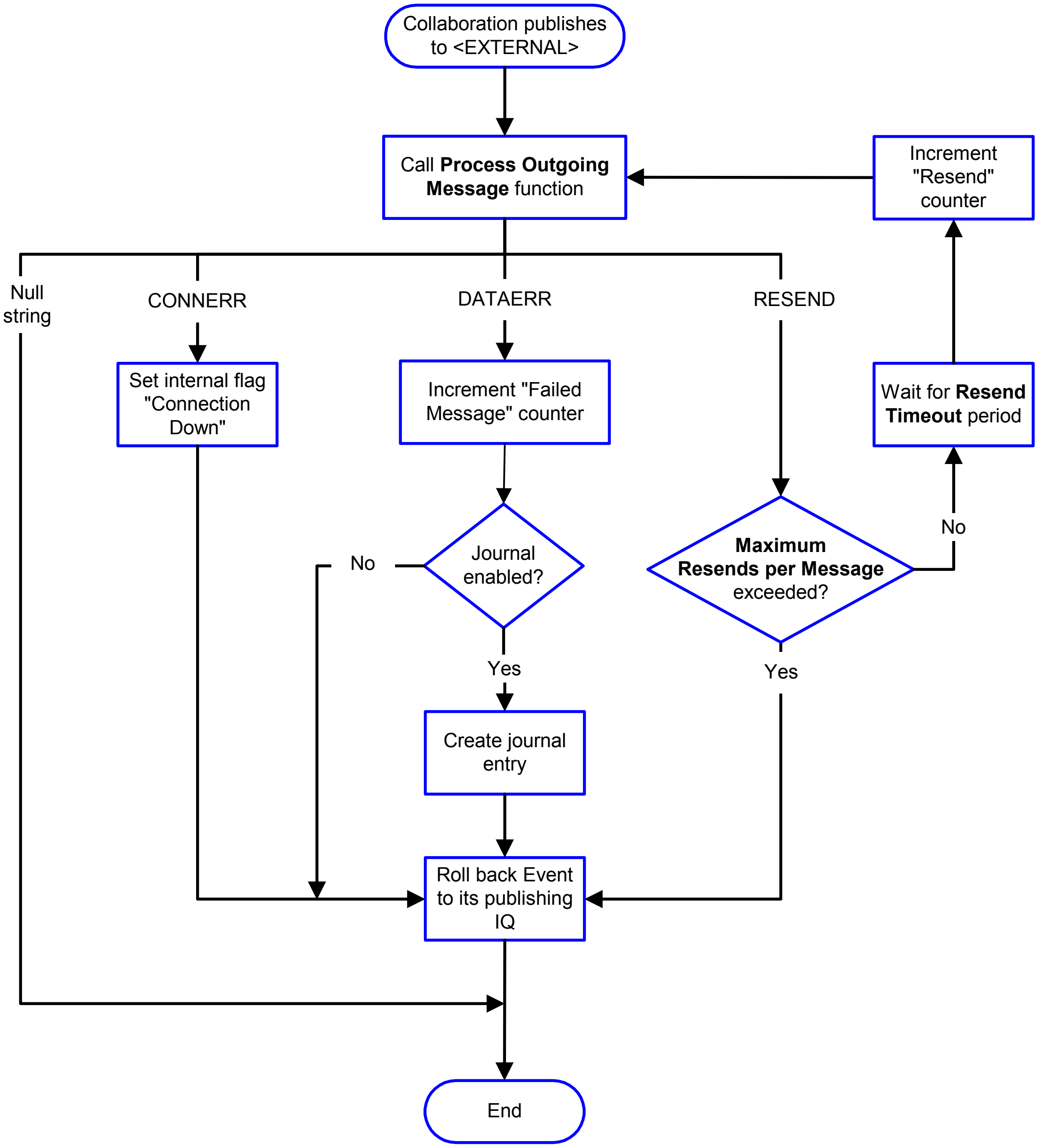

Control Broker issues "Shutdown" command
Call **Shutdown Notification** function with parameter "SHUTDOWN_NOTIFICATION"
Return
Null string or "SUCCESS"
any other value
Wait for **shutdown-request** function
e*Way shuts down

### Event-driven Data Exchange Functions

Figure 8 on the next page illustrates event-driven data-exchange using the **Process Outgoing Message Function**.

Every two minutes, the e*Way checks the “Failed Message” counter against the value specified by the **Max Failed Messages** parameter. When the “Failed Message” counter exceeds the specified maximum value, the e*Way logs an error and shuts down.

**Figure 8** Event-driven data-exchange functions


Collaboration publishes to <EXTERNAL>
Call **Process Outgoing Message** function
Increment "Resend" counter
Null string
CONNERR
DATAERR
RESEND
Set internal flag "Connection Down"
Increment "Failed Message" counter
Wait for **Resend Timeout** period
No
Journal enabled?
**Maximum Resends per Message** exceeded?
No
Yes
Yes
Create journal entry
Roll back Event to its publishing IQ
End

## How to Specify Function Names or File Names

Parameters that require the name of a Monk function will accept either a function name or a file name. If you specify a file name, be sure that the file has one of the following extensions:

- .monk
- .tsc
- .dsc

## Additional Path

### Description

Specifies a path to be added to the “load path,” the path Monk uses to locate files and data (set internally within Monk). The directory specified in Additional Path will be searched before the default load path.

### Required Values

A pathname, or a series of paths separated by semicolons. This parameter is optional and may be left blank.

### Additional information

The default load paths are determined by the “bin” and “Shared Data” settings in the .egate.store file. See the *e*Gate Integrator System Administration and Operations Guide* for more information about this file.

To specify multiple directories, manually enter the directory names rather than selecting them with the “file selection” button. Directory names must be separated with semicolons, and you can mix absolute paths with relative e*Gate paths. For example:

```
monk_scripts\my_dir;c:\my_directory
```

The internal e*Way function that loads this path information is called only once, when the e*Way first starts up.

## Auxiliary Library Directories

### Description

Specifies a path to auxiliary library directories. Any .monk files found within those directories will automatically be loaded into the e*Way’s Monk environment.

### Required Values

A pathname, or a series of paths separated by semicolons. (The default is **monk_library/dart**.)

### Additional information

To specify multiple directories, manually enter the directory names rather than selecting them with the “file selection” button. Directory names must be separated with semicolons, and you can mix absolute paths with relative e*Gate paths. For example:

```
monk_scripts\my_dir;c:\my_directory
```

The internal e*Way function that loads this path information is called only once, when the e*Way first starts up.

This parameter is optional and may be left blank.

## Monk Environment Initialization File

Specifies a file that contains environment initialization functions, which will be loaded after the auxiliary library directories are loaded. Use this feature to initialize any global Monk variables that are used by the Monk Extension scripts.

### Required Values

A filename within the “load path”, or filename plus path information (relative or absolute). If path information is specified, that path will be appended to the “load path.” See **Additional Path** on page 39 for more information about the “load path.”(The default is **db-stdver-init** on page 97**.**)

### Additional information

Any environment-initialization functions called by this file accept no input, and must return a string. The e*Way will load this file and try to invoke a function of the same base name as the file name (for example, for a file named **my-init.monk**, the e*Way would attempt to execute the function **my-init**).

Typically, it is a good practice to initialize any global Monk variables that may be used by any other Monk Extension scripts.

The internal function that loads this file is called once when the e*Way first starts up (see **Figure 3 on page 33**).

## Startup Function

### Description

Specifies a Monk function that the e*Way will load and invoke upon startup or whenever the e*Way’s configuration changes before it enters into its initial communication state. This function is used so that the external system can be initialized before message exchange starts.

### Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function.(The default is **db-stdver-startup** on page 105.)

### Additional information

The function accepts no input, and must return a string.

The string “FAILURE” indicates that the function failed; any other string (including a null string) indicates success.

This function will be called after the e*Way loads the specified “Monk Environment Initialization file” and any files within the specified **Auxiliary Directories**.

The e*Way will load this file and try to invoke a function of the same base name as the file name (see **Figure 3 on page 33**). For example, for a file named **my-startup.monk**, the e*Way would attempt to execute the function **my-startup**.

## Process Outgoing Message Function

### Description

Specifies the Monk function responsible for sending outgoing messages (Events) from the e*Way to the external system. This function is event-driven (unlike the Exchange Data with External function, which is schedule-driven).

### Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function. ***You may not leave this field blank.*** (The default is **db-stdver-proc-outgoing** on page 100 or **db-stdver-proc-outgoing-stub** on page 102**.**)

### Additional Information

The function requires a non-null string as input (the outgoing Event to be sent) and must return a string.

The e*Way invokes this function when one of its Collaborations publishes an Event to an <EXTERNAL> destination (as specified within the Schema Designer). The function returns one of the following (see **Figure 8 on page 38** for more details):

- Null string: Indicates that the Event was published successfully to the external system.
- “RESEND”: Indicates that the Event should be resent.
- “CONNERR”: Indicates that there is a problem communicating with the external system.
- “DATAERR”: Indicates that there is a problem with the message (Event) data itself.
- If a string other than the following is returned, the e*Way will create an entry in the log file indicating that an attempt has been made to access an unsupported function.

***Note:*** *If you wish to use* ***event-send-to-egate*** *to enqueue failed Events in a separate IQ, the e*Way must have an inbound Collaboration (with appropriate IQs) configured to process those Events. See* **event-send-to-egate** *on page 82 for more information.*

## Exchange Data with External Function

### Description

Specifies a Monk function that initiates an exchange of data with an external system.This function can exchange Events either inbound or outbound. This function is used with schedule based exchanges of data, predominantly inbound.

### Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function. (The defaults are **db-stdver-data-exchg** on page 95 or **db-stdver-data-exchg-stub** on page 96.)

### Additional Information

The function accepts no input and must return a string (see **Figure 6 on page 36** for more details):

- Null string: Indicates that the data exchange was completed successfully. No information will be sent into the e*Gate system.
- “CONNERR”: Indicates that a problem with the connection to the external system has occurred.

- "DATAERR": Indicates that a problem with the data itself has occurred. The e*Way handles the string "DATAERR" and "DATAERR" plus additional data differently; see **Figure 6 on page 36** for more details.
- Any other string: The contents of the string are packaged as an inbound Event. The e*Way must have at least one Collaboration configured suitably to process the inbound Event, as well as any required IQs.

This function is initially triggered by the **Start Data Exchange** schedule or manually by the Monk function **start-schedule**. After the function has returned true and the data received by this function has been ACKed or NAKed (by the **Positive Acknowledgment Function** or **Negative Acknowledgment Function**, respectively), the e*Way checks the **Zero Wait Between Successful Exchanges** parameter. If this parameter is set to **Yes**, the e*Way will immediately call the **Exchange Data with External** function again; otherwise, the e*Way will not call the function until the next scheduled "start exchange" time or the schedule is manually invoked using the Monk function **start-schedule** (see **start-schedule** on page 87 for more information).

## External Connection Establishment Function

### Description

Specifies a Monk function that the e*Way will call to establish (or re-establish) a connection to the external system.

### Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function. ***This field cannot be left blank***. (The default is **db-stdver-conn-estab** on page 90.)

### Additional Information

The function accepts no input and must return a string:

- "SUCCESS" or "UP": Indicates that the connection was established successfully.
- Any other string (including the null string): Indicates that the attempt to establish the connection failed.

This function is executed according to the interval specified within the **Down Timeout** parameter, and is *only* called according to this schedule.

The **External Connection Verification** function (see below) is called when the e*Way has determined that its connection to the external system is up.

## External Connection Verification Function

### Description

Specifies a Monk function that the e*Way will call to confirm that the external system is operating and available.

### Required Values

The name of a Monk function. This function is optional; if no **External Connection Verification** function is specified, the e*Way will execute the **External Connection Establishment** function in its place. (The default is **db-stdver-conn-ver** on page 93.)

### Additional Information

The function accepts no input and must return a string:

- "SUCCESS" or "UP": Indicates that the connection was established successfully.
- Any other string (including the null string): Indicates that the attempt to establish the connection failed.

This function is executed according to the interval specified within the **Up Timeout** parameter, and is *only* called according to this schedule.

The **External Connection Establishment** function (see above) is called when the e*Way has determined that its connection to the external system is down.

## External Connection Shutdown Function

### Description

Specifies a Monk function that the e*Way will call to shut down the connection to the external system.

### Required Values

The name of a Monk function. (The default is **db-stdver-conn-shutdown** on page 92.)

### Additional Information

This function requires a string as input, and may return a string.

This function will only be invoked when the e*Way receives a "suspend" command from a Control Broker. When the "suspend" command is received, the e*Way will invoke this function, passing the string "SUSPEND_NOTIFICATION" as an argument.

Any return value indicates that the "suspend" command can proceed and that the connection to the external system can be broken immediately.

***Note:*** *Include in this function any required "clean up" that must be performed as part of the shutdown procedure, but before the e*Way exits.*

## Positive Acknowledgment Function

### Description

Specifies a Monk function that the e*Way will call when ***all*** the Collaborations to which the e*Way sent data have processed and enqueued that data successfully.

### Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function. This parameter is required if the **Exchange Data with External** function is defined. (The default is **db-stdver-pos-ack** on page 99.)

### Additional Information

The function requires a non-null string as input, and returns a string.

The function requires a non-null string as input (the Event to be sent to the external system) and must return a string:

- "CONNERR": Indicates a problem with the connection to the external system. When the connection is re-established, the Positive Acknowledgment function will be called again, with the same input data.
- Null string: The function completed execution successfully.

After the **Exchange Data with External** function returns a string that is transformed into an inbound Event, the Event is handed off to one or more Collaborations for further processing. If the Event's processing is completed successfully by *all* the Collaborations to which it was sent, the e*Way executes the Positive Acknowledgment function (otherwise, the e*Way executes the Negative Acknowledgment function).

**Note:** *If you configure the acknowledgment function to return a non-null string, you must configure a Collaboration (with appropriate IQs ) to process the returned Event.*

## Negative Acknowledgment Function

### Description

Specifies a Monk function the e*Way will call when the e*Way fails to process and queue data from the external system.

### Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function. This parameter is required if the **Exchange Data with External** function is defined. (The is default is **db-stdver-neg-ack** on page 98.)

### Additional Information

The function requires a non-null string as input (the Event to be sent to the external system) and must return a string:

- "CONNERR": Indicates a problem with the connection to the external system. When the connection is re-established, the function will be called again.
- Null string: The function completed execution successfully.

This function is only called during the processing of inbound Events. After the **Exchange Data with External** function returns a string that is transformed into an inbound Event, the Event is handed off to one or more Collaborations for further processing. If the Event's processing is not completed successfully by *all* the Collaborations to which it was sent, the e*Way executes the Negative Acknowledgment function (otherwise, the e*Way executes the Positive Acknowledgment function).

**Note:** *If you configure the acknowledgment function to return a non-null string, you must configure a Collaboration (with appropriate IQs ) to process the returned Event.*

The e*Way will exit if it fails its attempt to invoke this function or this function returns a **FAILURE** string.

## Shutdown Command Notification Function

### Description

Specifies a Monk function that will be called when the e*Way receives a "shut down" command from the Control Broker. This parameter is optional.

### Required Values

The name of a Monk function. (The default is **db-stdver-shutdown** on page 104.)

### Additional Information

When the Control Broker issues a shutdown command to the e*Way, the e*Way will call this function with the string "SHUTDOWN_NOTIFICATION" passed as a parameter.

The function accepts a string as input and must return a string:

- A null string or "SUCCESS": Indicates that the shutdown can occur immediately.
- Any other string: Indicates that shutdown must be postponed. Once postponed, shutdown will not proceed until the Monk function **shutdown-request** is executed.

***Note:*** *If you postpone a shutdown using this function, be sure to use the (**shutdown-request)** function to complete the process in a timely manner.*

## 3.2.4 Database Setup

## Database Type

### Description

Specifies the type of database.

### Required Values

**DB2**, **ODBC**, **ORACLE7**, **ORACLE8**, **ORACLE8i**, **SYBASE11**, or **SYBASE12**

***Note:*** *Any other value is effectively equal to ODBC.*

## Database Name

### Description

The name of the database.

### Required Values

None. Any valid string.

## User Name

### Description

The name used to access the database.

### Required Values

None. Any valid string.

## Encrypted Password

### Description

The password that provides access to the database.

### Required Values

Any valid string.

***Note:*** *Changes to Monk files can be made using the Collaboration Rules Editor (available from within the Schema Designer) or with a text editor. However, if you use a text editor to edit Monk files directly, you **must** commit these changed files to the e*Gate Registry or your changes will not be implemented.*

*For more information about committing files to the e*Gate Registry, see the Schema Designer's online Help system, or the "stcregutil" command-line utility in the **e*Gate Integrator System Administration and Operations Guide**.*

Chapter 4

# Implementation

This chapter contains information explaining the use of the ETD Editor's Build Tool as well as two sample ODBC e*Way scenarios.

**This Chapter Explains:**

## 4.1 Using the ETD Editor's Build Tool

The Event Type Definition Editor's Build tool automatically creates an Event Type Definition file based on the tables in an existing database. The Event Type Definition (ETD) can be created based on one of (or a combination of) the following criteria:

- **Table or View** – Displays all of the columns in the specified table or view.
- **Dynamic SQL Statement** – Displays the format of the results of a SQL statement. This can be used to return only a few of the columns in a table.
- **Stored Procedure** – Displays the format of the results of a SQL Stored Procedure. This option is only available for *Delimited* messages.

The results of these three types of message criteria are explained in **"The Event Type Definition Files" on page 50**.

**To create an Event Type Definition using the Build tool:**

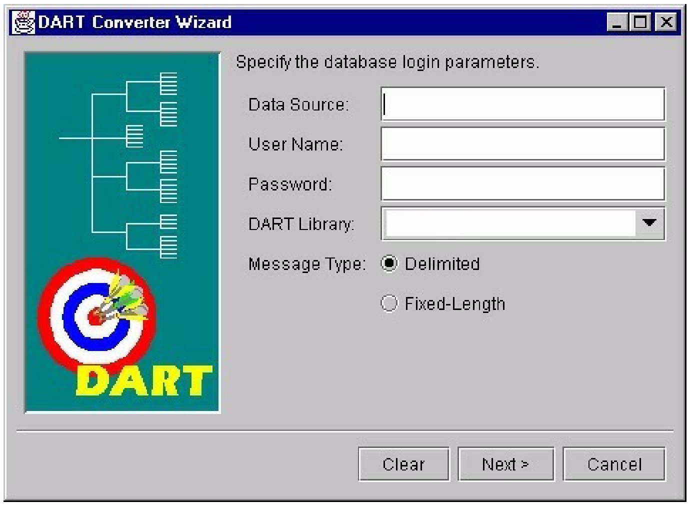
1. Launch the ETD (Event Type Definition) Editor.
2. On the ETD Editor's Toolbar, click **Build**.

   The **Build an Event Type Definition** dialog box appears.
3. In the File name box, type the name of the ETD file you wish to build. *Do not specify any file extension*—the Editor will supply an "ssc" extension for you.
4. Under **Build From**, select **Library Converter**.
5. Under **Select a Library Converter**, select DART Converter.

6 Click **Finish**.

7 The Converter Wizard will launch.

**Figure 9** Converter Wizard Subordinate Dialog Box



Specify the database login parameters.

Data Source:

User Name:

Password:

DART Library:

Message Type: ● Delimited ○ Fixed-Length

8 Enter the Data Source.

9 Enter the User Name.

10 Enter the Password.

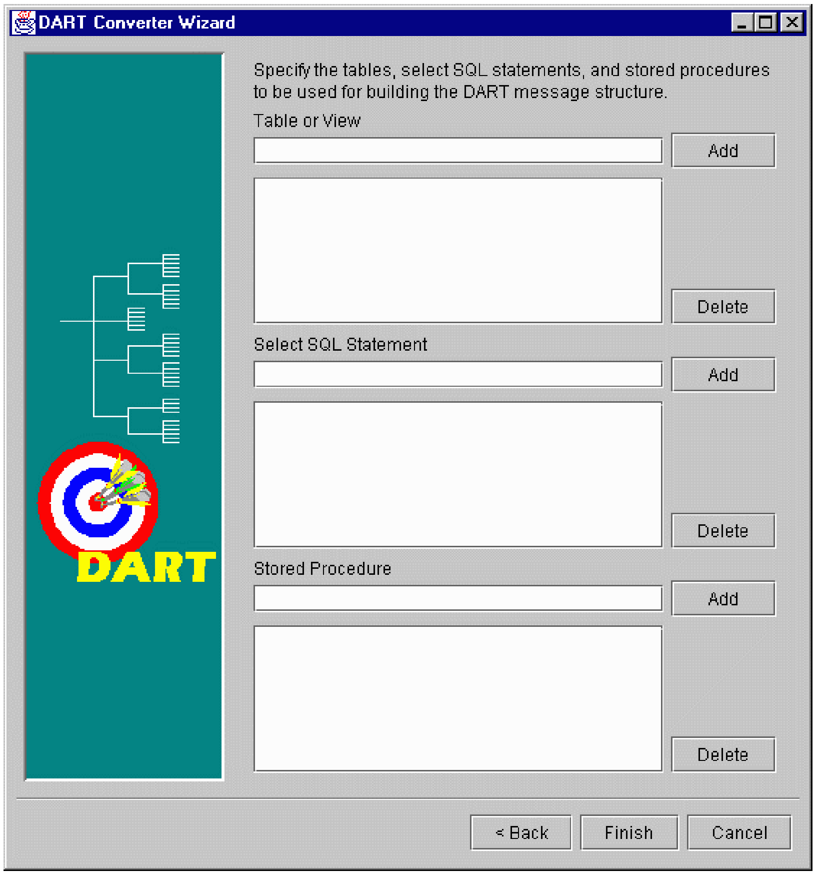
11 Select the DART Library. You must have the corresponding e*Way installed prior to your selection.

12 Select the correct Message Type.

***Note:*** *It is important to enter the correct Data Source and Message Type. For Oracle the Data Source is in the Servicename.world format*
*The Fixed-length Message Type is used for DART bulk insert only.*
*The Delimited Message Type is for all other DART structure calls.*
*See* **Figure 9 on page 48**

If you select Delimited Message Type the following dialog box will appear.

**Figure 10** Converter Wizard Delimited Message Type Dialog Box



DART Converter Wizard

Specify the tables, select SQL statements, and stored procedures to be used for building the DART message structure.

Table or View

Select SQL Statement

Stored Procedure

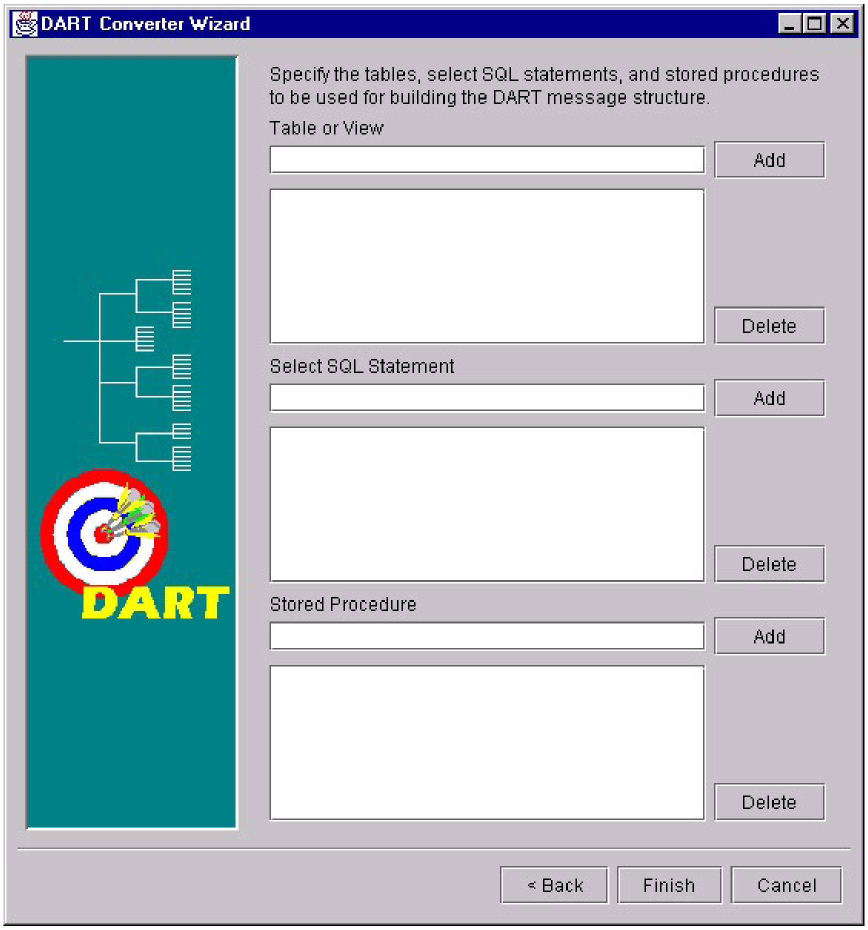
13 Select or Add the correct Table or View

14 Select or Add the correct SQL Statement

15 Select or Add the correct Stored Procedure.

If you select the Fixed-Length Message Type the following dialog box will appear.

**Figure 11** Converter Wizard Fixed-Length Message Type Dialog Box



Specify the tables, select SQL statements, and stored procedures to be used for building the DART message structure.

16 Select or Add the correct Table or View

17 Select or Add the correct SQL Statement

18 Edit or Finish your selections.

***Note:*** *The (#) character cannot be used in the node name of the .ssc file. The Oracle e*Way will be unable to generate the correct node name for the column name of a table that contains the (#) character, as Monk will filter out the character.*

For Oracle, ($), or (#) can be used in a name, although the Oracle User's Guide strongly discourages their use.

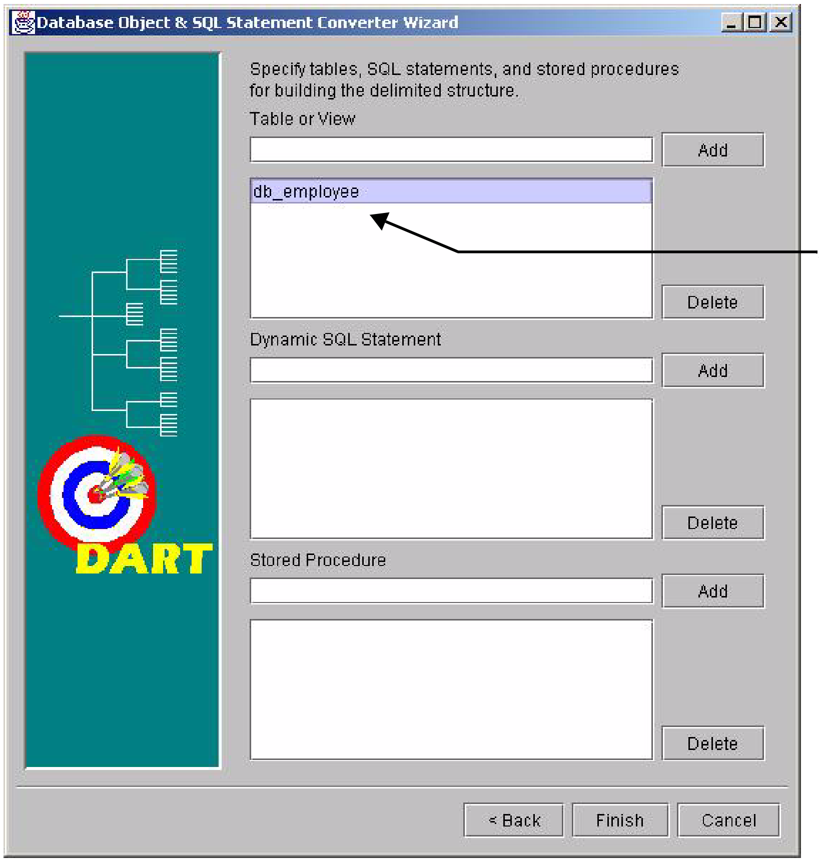
## 4.1.1 The Event Type Definition Files

The DART Converter Build Tool will create a different ETD based on the criteria that was specified in the Build Tool Wizard (see **Figure 10 on page 49** and **Figure 11 on page 50**).

## Table or View

Entering a table or view name as a selection criteria will display all of the columns in that table or view. This is useful when you want to access an entire record from the table as an e*Gate Event. The criteria shown in Figure 12 generates the ETD shown in Figure 13.

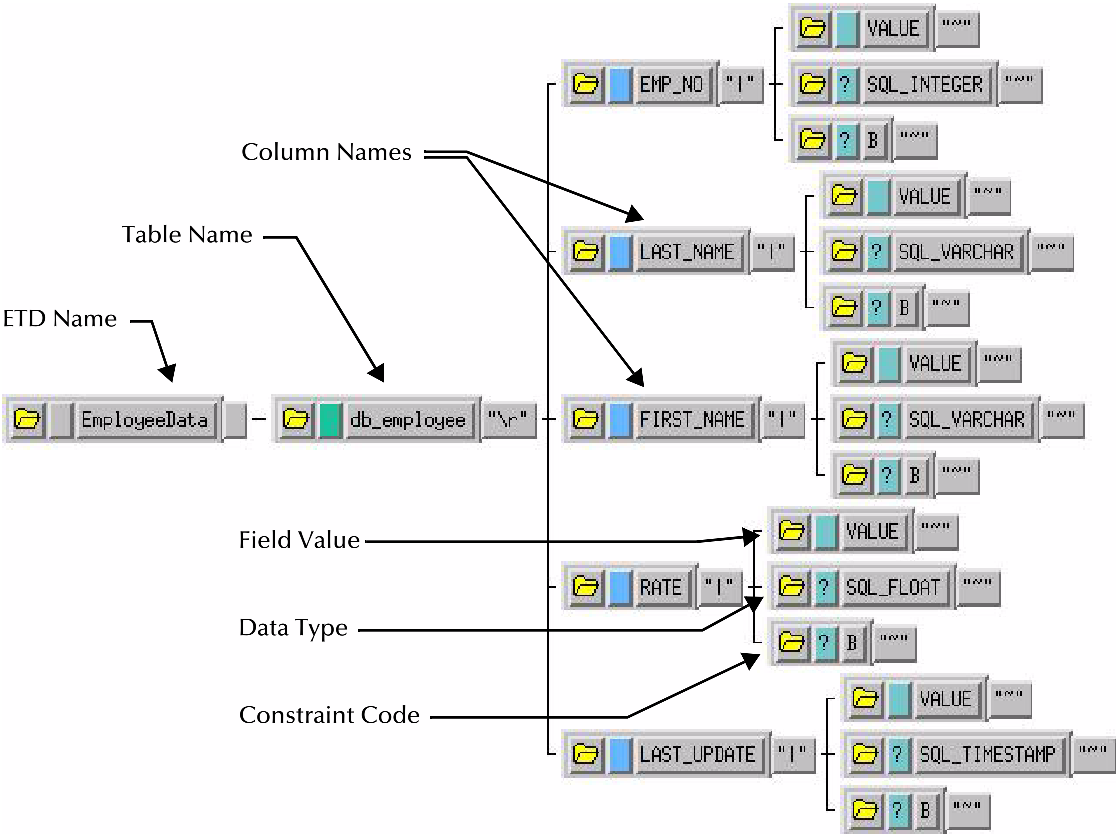
**Figure 12** Table or View Selection



Database Object & SQL Statement Converter Wizard
Specify tables, SQL statements, and stored procedures for building the delimited structure.
Table or View
Add
db_employee
Delete
Dynamic SQL Statement
Add
Delete
Stored Procedure
Add
Delete
DART
< Back
Finish
Cancel

Selection criteria

**Figure 13** Table or View ETD



The ETD that is generated by the DART Converter Build Tool using the Table or View criteria contains the elements shown in the table below.

**Table 3** Elements of the Table or View ETD

| Element | Description |
|---|---|
| ETD Name | This is the root node of the Event Type Definition. |
| Table Name | This node displays the name of the table or view. |
| Column Name | This is the name of the column(s) in the selected table or view. |
| Field Value | This is the value of the data in the column. This can be thought of as the *payload data* for this column. |
| Data Type | This node designates the type of data contained in the value field. |
| Constraint Code | The constraint codes are based on the column constraints in the table. The possible codes are:<br>▪ **I** – *Insert* operations are allowed in this column.<br>▪ **U** – *Update* operations are allowed in this column.<br>▪ **N** – *Neither* insert nor update operations are allowed in this column.<br>▪ **B** – *Both* insert and update operations are allowed in this column. |

## Dynamic SQL Statement

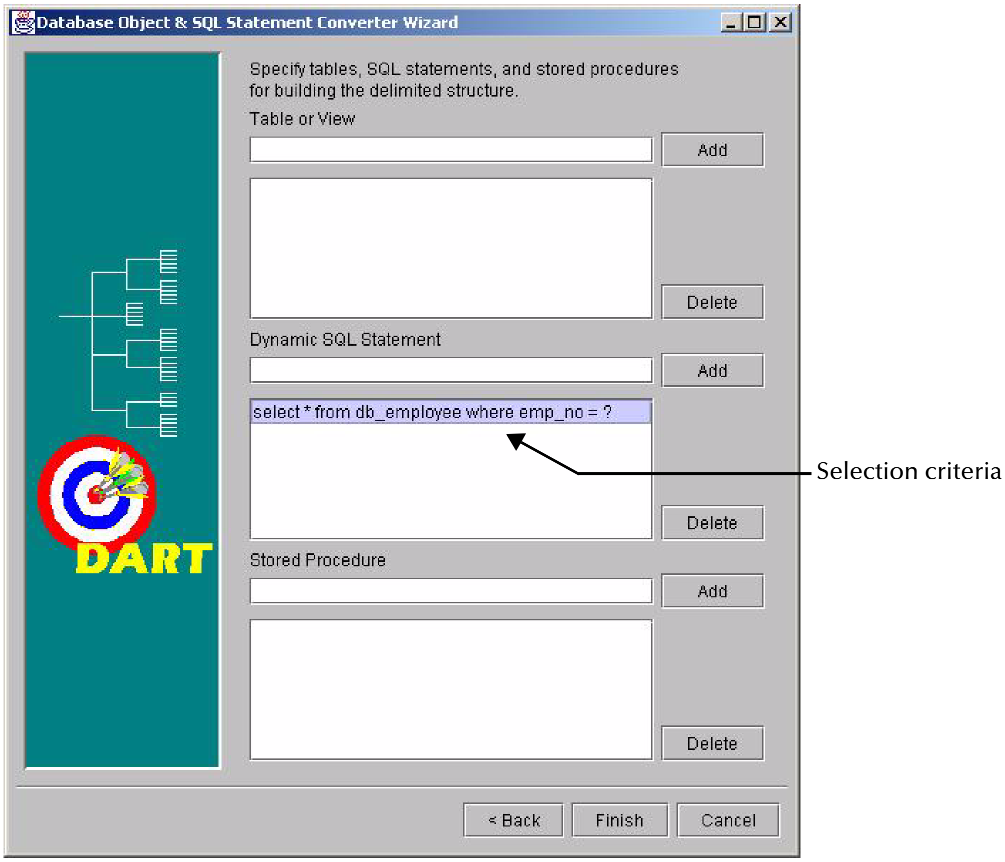
Entering an SQL statement as a selection criteria will display the format of the results of that SQL statement. This is useful when you only want to access certain columns from the table for a particular e*Gate Event.

To use this type of ETD, you should use the **db-stmt-bind** function to bind the dynamic statement and **db-struct-execute** function to execute the SQL statement. For more information, see **db-stmt-bind** on page 138 and **db-struct-execute** on page 180.

The SQL statement shown in Figure 14 generates an ETD that returns specific records from the table based on the selection criteria (which is represented by a question mark "**?**"). The resulting ETD is shown in **Figure 15 on page 54**.

***Note:*** *It is not necessary to include the terminating semi-colon as part of the SQL statement.*

**Figure 14** Dynamic SQL Statement Select

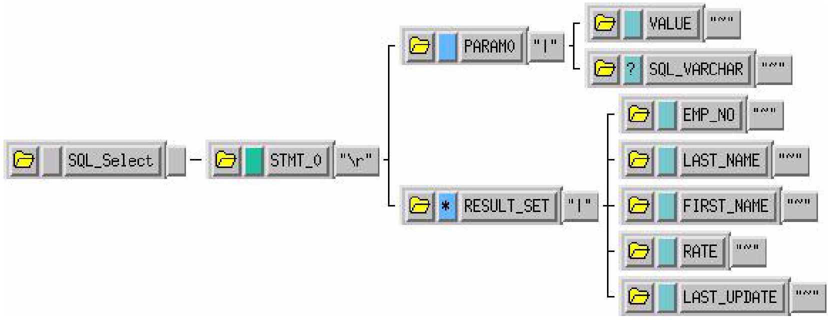


Specify tables, SQL statements, and stored procedures for building the delimited structure.

Dynamic SQL Statement

select * from db_employee where emp_no = ?

**Figure 15** Dynamic SQL Statement ETD



The **PARAM0** node in the ETD shown in Figure 15 represents the criteria specified in the SQL statement. Additional criteria would be represented in additional nodes (**PARAM1**, **PARAM2**, and so forth). For example, using the following SQL statement:

```
SELECT * FROM db_employee WHERE last_name = ? AND first_name = ?
```

the Build Tool would generate an ETD with two input parameter nodes (**PARAM0** and **PARAM1**)—one for each of the criteria (**?**). The **VALUE** nodes of these input parameter nodes are used to carry the payload of the selection statement.

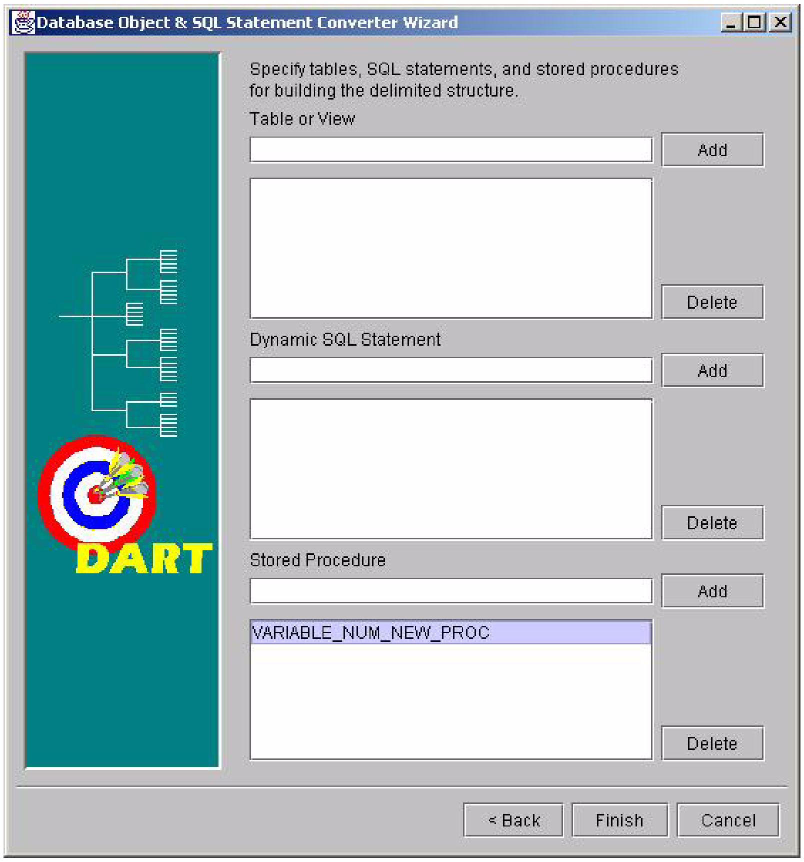
## Stored Procedure

Entering a stored procedure name as a selection criteria will generate an ETD that will access a stored procedure in the external database. This is useful when you want to access the results of a stored procedure.

The stored procedure specified in Figure 16 generates an the ETD shown in Figure 17. Below is the contents of the sample stored procedure:

```
procedure VARIABLE_NUM_NEW_PROC
(
    BATCH_SIZE          in integer,
    FOUND               in out integer,
    DONE_FETCH          out integer,
    INT_RET             out integer,
    FLOAT_RET           out float,
    SMALL_INT_RET       out smallint,
    DOUBLE_RET          out double precision,
    REAL_RET            out real,
    DECIMAL_RET         out decimal,
    DECIMAL_PRECISE_RET out decimal,
    NUMBER_RET          out number,
    NUMBER_PRECISE_RET  out number
)
as
  temp int := 0;
  cursor GET_COUNT IS
    select count(*) from NUM_TABLE;
  cursor GET_TYPE IS
   select INT_NUM, FLOAT_NUM, SMALL_INT_NUM, DOUBLE_NUM, REAL_NUM,
        DECIMAL_NUM, DECIMAL_PRECISE, NUM_NUM, NUM_PRECISE
     from NUMBER_TYPE;
begin
  OPEN GET_COUNT;
    fetch GET_COUNT into temp;
  CLOSE GET_COUNT;
  DONE_FETCH := temp;
  if temp = 0
  then
    FOUND := 0;
  else
    FOUND := 1;
  end if;
  OPEN GET_TYPE;
    fetch GET_TYPE into INT_RET, FLOAT_RET, SMALL_INT_RET, DOUBLE_RET,
        REAL_RET, DECIMAL_RET, DECIMAL_PRECISE_RET, NUMBER_RET,
        NUMBER_PRECISE_RET;
  CLOSE GET_TYPE;
end;
```

**Figure 16** Stored Procedure Selection



Specify tables, SQL statements, and stored procedures for building the delimited structure.

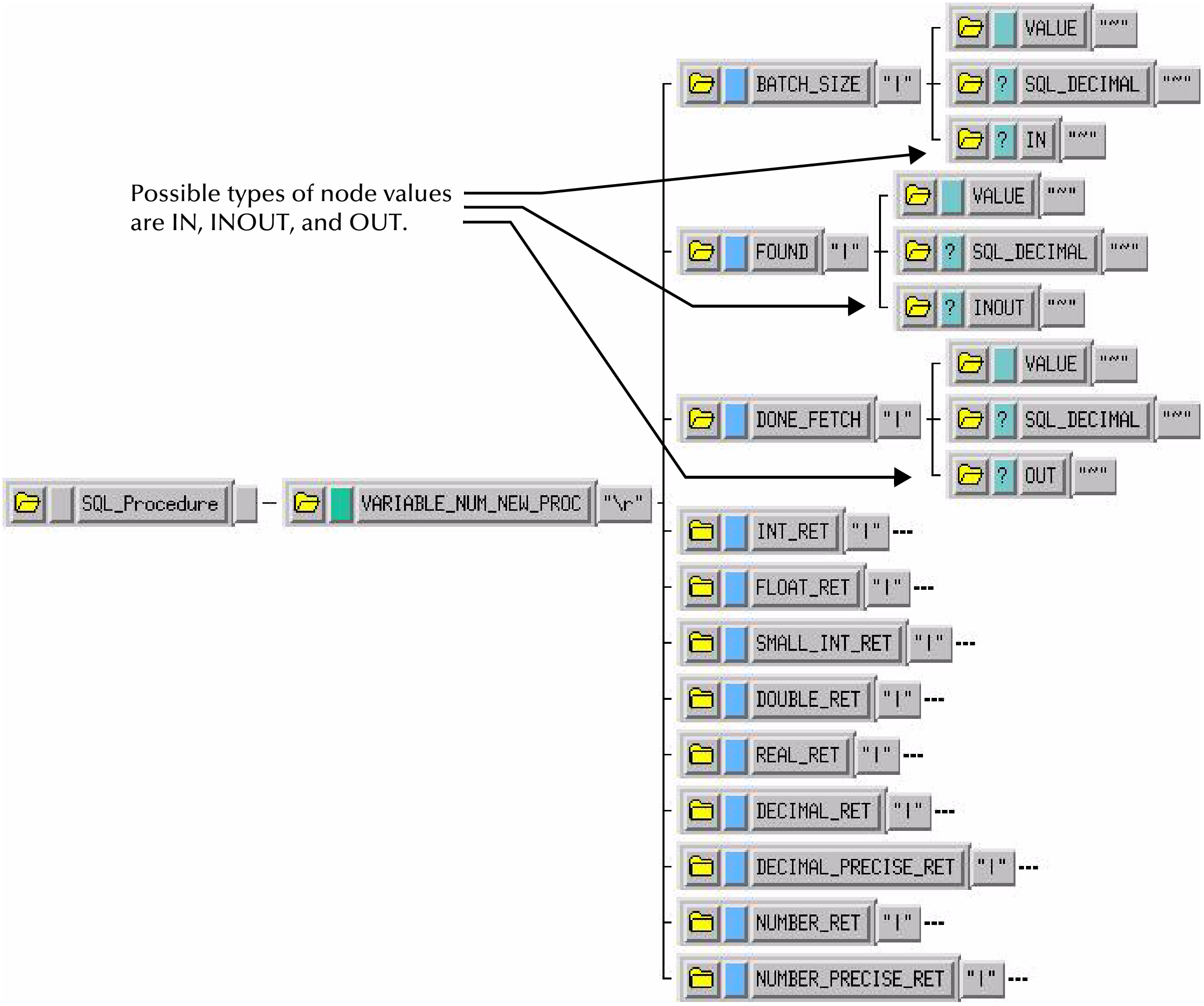
Table or View

Dynamic SQL Statement

Stored Procedure

VARIABLE_NUM_NEW_PROC

**Figure 17** Stored Procedure ETD



Possible types of node values are IN, INOUT, and OUT.

This Event Type Definition is used to pass certain input to the stored procedure. The nodes with types of "IN" or "INOUT" are used as input. The results are returned to the "OUT" or "INOUT" nodes.

## 4.2 Vendor-Specific Driver Notes

Certain functions are known to behave differently based on the type of ODBC drivers being used on the client machine. These differences in functionality are explained below.

### 4.2.1 IBM ODBC DB2 Drivers

The following issues are known to exist with the IBM DB2 ODBC drivers.

### Support for BLOB and CLOB Data Types

The IBM ODBC DB2 drivers support BLOB (Binary Large Object) or CLOB (Character Large Object) data only if specifically configured to do so.

**Note:** *Large records (CLOB or BLOB) should be inserted by using the* ***db-stmt-bind*** *and* ***db-stmt-execute*** *functions. For more information on these functions, see* **"db-stmt-bind" on page 138** *and* **"db-stmt-execute" on page 143**.

Follow the procedures below for configuring your client workstation to support BLOB and CLOB data.

#### Configuring a Windows Client to Support Long Objects

1. Open the **Control Panel**.
2. Open the **Administrative Tools** (Windows 2000 only).
3. Open the **ODBC Data Sources**.
4. Select the **System DSN** tab.
5. Select appropriate driver for your IBM DB2 implementation (such as "IBM_UDB2") and click **Configure**.
6. Enter the appropriate username and password to connect to the DB2 data source.
7. Select the **Data Type** tab.
8. Select the **Long object binary treatment** parameter.
9. Choose the **As LONGVAR data** setting.
10. Save the settings and close the ODBC configuration utility.

#### Configuring a UNIX Client to Support Long Objects

1. Use a text editor to open the **db2cli.ini** file.
2. Add the following line:

```
LONGDATACOMPAT = 1
```

**Note:** *Large records using the CLOB data type are limited to approximately 1GB in size. Larger records should not be inserted.*

## 4.2.2 Merant ODBC Drivers

The following issues are known to exist with the Merant ODBC drivers.

### Support for BLOB and CLOB Data Types

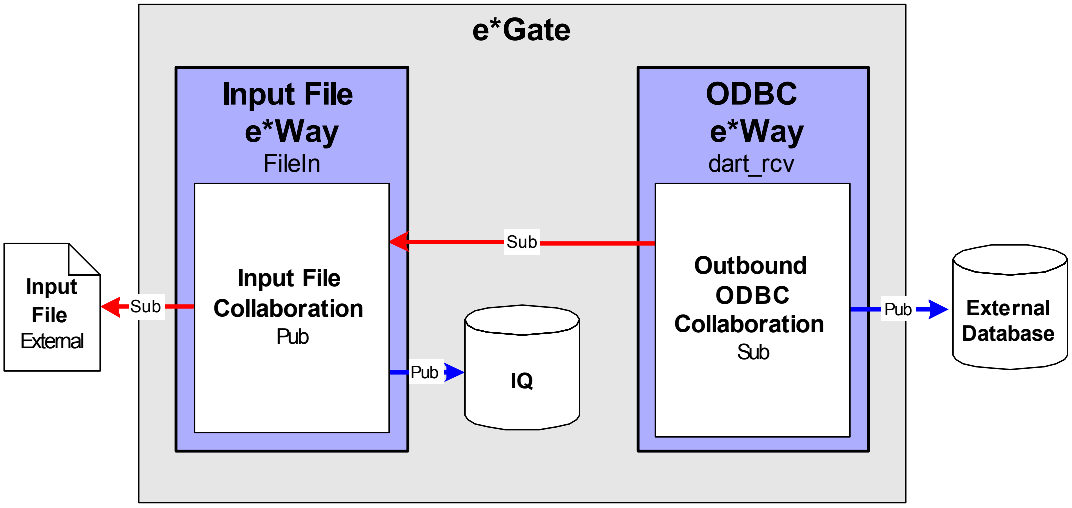
The Merant ODBC drivers do not support BLOB (Binary Large Object) or CLOB (Character Large Object) data. This affects the size and type of large records that can be inserted into tables from a client using the Merant ODBC drivers.

# 4.3 Sample One—Publishing e*Gate Events to an ODBC Database

This section describes how to use the ODBC e*Way in a sample implementation. This sample schema demonstrates the publishing of e*Gate Events to an external database.

This scenario uses a file e*Way to load an input file containing employee information and generate the initial Event. The ODBC e*Way subscribes to the Event and inserts the employee records into the external database.

**Figure 18** Publishing to an External Database

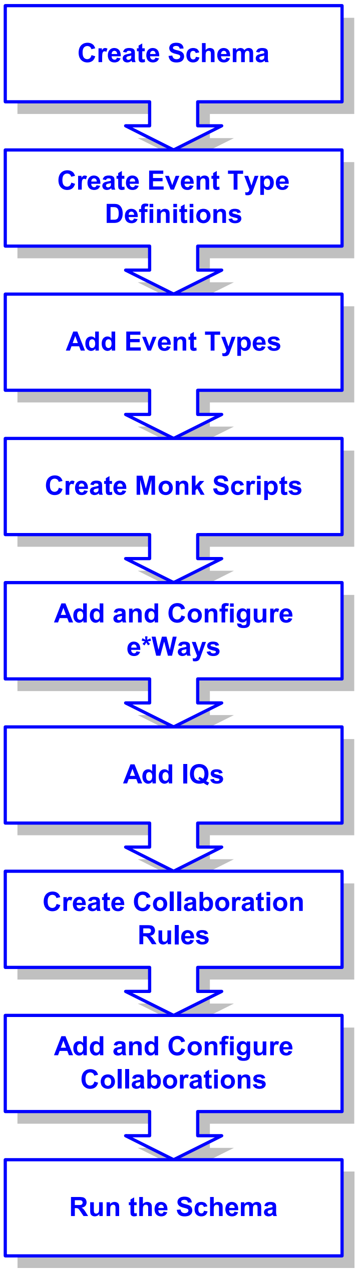


### Overview of Steps

The sample implementation follows these general steps:

- “Create the Schema” on page 59
- “Create the Event Type Definitions” on page 60
- “Add the Event Types” on page 61
- “Create the Monk Scripts” on page 62
- “Add and Configure the e*Ways” on page 62
- “Add the IQs” on page 64
- “Create the Collaboration Rules” on page 65
- “Add and Configure the Collaborations” on page 66
- “Run the Schema” on page 67

**Figure 19** Schema Configuration Steps



## 4.3.1 Create the Schema

The first step in deploying the sample implementation is to create a new Schema. After installing the ODBC e*Way Intelligent Adapter, do the following:

1. Launch the e*Gate Schema Designer GUI.
2. When the Schema Designer prompts you to log in, select the Registry Host, User Name, and Password to be used to log in and click **Log In**.
3. From the list of Schemas, click **New** to create a new Schema.

4 For this sample implementation, enter the name **ODBC_Sample1** and click **Open**.

The Schema Designer will launch and display the newly created Schema.

## 4.3.2 Create the Event Type Definitions

Three Event Type Definitions are used in this sample. The ETDs are:

- **EventMsg.ssc** – This standard ETD is used by the **FileInEvent** Event Type.
- **db_rcv_in.ssc** – This user-created ETD contains basic employee information such as name, rate, and date.
- **db_rcv_struct.ssc** – This user-created ETD contains the same basic employee information formatted appropriately for the external data source.

**To create the db_rcv_in ETD:**

1 From the e*Gate Schema Designer, click to launch the ETD Editor.

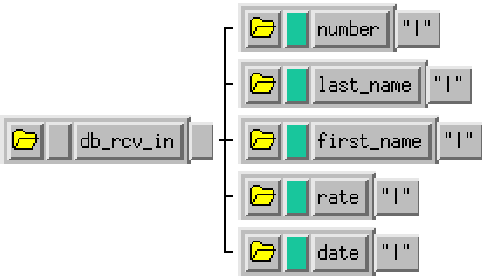
2 Click New to create the new ETD.

The New ETD dialog will be displayed.

3 Enter **db_rcv_in.ssc** as the file name for the ETD.

4 Add the nodes and subnodes to create an ETD with the structure shown below:

**Figure 20** The **db_rcv_in.ssc** ETD



5 Click Save to save the ETD.

6 From the **File** menu, select **Promote to Run Time**. Click **Yes** to confirm the promotion of the file.

**To create the db_rcv_struct ETD:**

1 From the e*Gate Schema Designer, click to launch the ETD Editor.

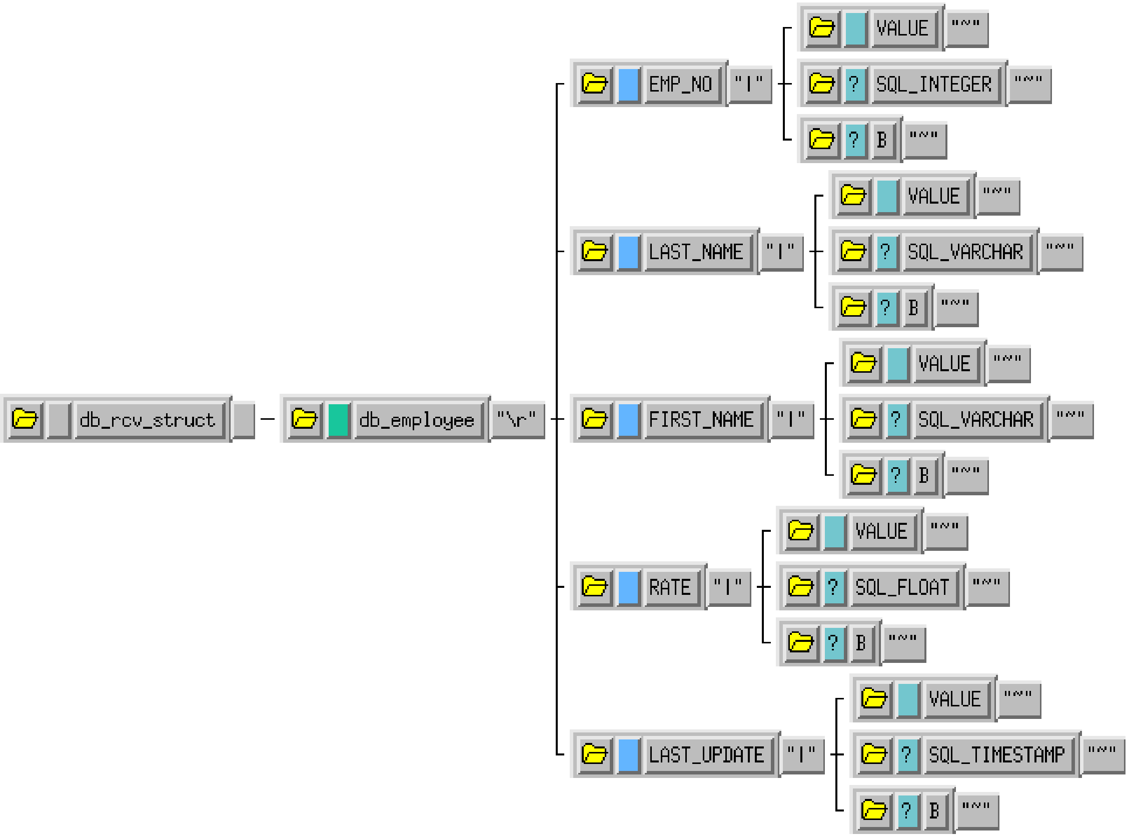
2 Click New to create the new ETD.

The New ETD dialog will be displayed.

3 Enter **db_rcv_struct.ssc** as the file name for the ETD.

4 Add the nodes and subnodes to create an ETD with the structure shown below:

**Figure 21** The **db_rcv_struct.ssc** ETD



5 Click  to save the ETD.

6 From the File menu, select **Promote to Run Time**. Click **Yes** to confirm the promotion of the file.

## 4.3.3 Add the Event Types

Three Event Types are used in this sample. The Event Types are:

- **FileInEvent** – This Event Type represents the inbound data from an external input file. This Event Type uses the **EventMsg.ssc** ETD.
- **db_rcv_in** – This Event Type represents the data transported by the input file e*Way. This Event Type uses the **db_rcv_in.ssc** ETD.
- **db_rcv_struct** – This Event Type represents the transformed Event that will be written to the external database. This Event Type uses the **db_rcv_struct.ssc** ETD.

**To add the Event Types:**

1 In the components pane of the Schema Designer, select the Event Types folder.

2 Click  to add a new Event Type.

3 Enter **FileInEvent** and click **OK**.

4. Select the newly created Event Type and click to display the Event Type's properties.
5. Click **Find** to display the list of Event Types.
6. Navigate to the **monk_scripts\common** folder, select **EventMsg.ssc**, and click **Select**.
7. Click **OK** to close the Event Type's properties.

   Repeat these steps for the **db_rcv_in** and **db_rcv_struct** Event Types using the appropriate Event Type Definition files.

## 4.3.4 Create the Monk Scripts

This sample implementation uses a DART script (**db_rcv.dsc**) to communicate with the external Oracle database.

**To create the DART script:**

1. From the e*Gate Schema Designer, click to launch the Collaboration Rules Editor.
2. Click New to create a new DART script.

   The New Collaboration Rules Script dialog will be displayed.
3. Enter the name **db_rcv** (with no file extension) as the **File name**.
4. Select **DART Send** from the list of file types. The extension **.dsc** will be appended to the file name.
5. Click to display the list of source files. Select **db_rcv_in.ssc** as the source file.
6. Click to display the list of destination files. Select **db_rcv_struct.ssc** as the destination file.
7. Enter the rules.
8. Click Save to save the script.
9. Close the Collaboration Rules Script Editor.

## 4.3.5 Add and Configure the e*Ways

The sample Schema uses two e*Ways: **FileIn** and **ODBC_rcv**. The **FileIn** e*Way reads in the input data file and queues it for the ODBC e*Way. The **ODBC_rcv** e*Way writes the records to the **db_employee** table in the external database.

**To add and configure the FileIn e*Way:**

1. In the components pane of the Schema Designer, select the Control Broker and click to add a new e*Way.

2 Enter **FileIn** for the component name and click **OK**.

3 Select the newly created e*Way and click 📋 to display the e*Way's properties.

4 Use the **Find** button to select **stcewfile.exe** as the executable file.

5 Click **New** to create a new configuration file.

6 Enter the parameters for the e*Way as shown in Table 4.

**Table 4** FileIn e*Way Parameters

| Section Name | Parameter | Value |
|---|---|---|
| General Settings | AllowIncoming | YES |
| | AllowOutgoing | NO |
| | PerformanceTesting | default |
| Outbound (send) settings | All | default |
| Poller (inbound) settings | PollDirectory | c:\egate\data\dart |
| | InputFileMask | *.dat |
| | All others | default |
| Performance Testing | All | default |

7 Select **Save** from the **File** menu. Enter **FileIn** as the file name and click **Save**.

8 Select **Promote to Run Time** from the **File** menu. Click **OK** to continue.

9 A message will notify you that the file has been promoted to run time. Click **OK** to close the e*Way configuration file editor.

10 In the **Start Up** tab of the e*Way properties, select the **Start automatically** check box.

11 Click **OK** to save the e*Way properties.

**To add and configure the ODBC_rcv e*Way:**

1 In the components pane of the Schema Designer, select the Control Broker and click 🖥 to add a new e*Way.

2 Enter **ODBC_rcv** for the component name and click **OK**.

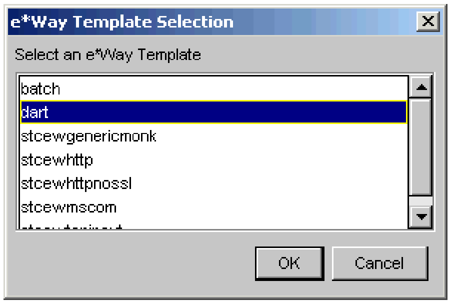
3 Select the newly created e*Way and click 📋 to display the e*Way's properties.

4 Use the **Find** button to select **stcewgenericmonk.exe** as the executable file.

5 Click **New** to create a new configuration file.

6 Select the **dart** e*Way template and click OK. See Figure 22.

**Figure 22** DART e*Way Template Selection



7 Enter the parameters for the e*Way as shown in Table 5.

**Table 5** ODBC_rcv e*Way Parameters

| Section Name | Parameter | Value |
|---|---|---|
| General Settings | All | default |
| Communication Setup | Start Exchange Data Schedule | Repeatedly, every 1 minute |
| | All others | default |
| Monk Configuration | Process Outgoing Message Function | monk_scripts\common\db_rcv.dsc |
| | Exchange Data With External Function | monk_scripts\common\db_rcv.dsc |
| | All others | default |
| Database Setup | Database Type | ODBC |
| | All others | Use local settings |

*Note: Use the appropriate* ***Database Name****,* ***User Name****, and* ***Encrypted Password*** *according to your local ODBC configuration.*

8 Save the e*Way's configuration file and promote it to run time.

9 In the **Start Up** tab of the e*Way properties, select the **Start automatically** check box.

10 Click **OK** to save the e*Way properties.

## 4.3.6 Add the IQs

The sample Schema requires one Intelligent Queue—**ODBC_IQ**.

**To add the IQ:**

1 In the components pane of the Schema Designer, select the IQ manager. Click  to create the new IQ.

2 Enter the name **ODBC_IQ** and click **OK** to save the IQ.

3 Select the IQ Manager and click  to display the IQ Manager's properties.

4 In the **Start Up** tab of the IQ Manager's properties, select the **Start automatically** check box.

5 Click **OK** to save the IQ Manager's properties.

## 4.3.7 Create the Collaboration Rules

This sample schema uses two Collaboration Rules:

- **InboundEvent** – This Collaboration Rule is used by the **FileIn** e*Way's collaboration to transform the **FileInEvent** Events into **db_rcv_in** Events.
- **OutboundEvent** – This Collaboration Rule is used by the **ODBC_rcv** e*Way's collaboration to transform the **db_rcv_in** Events into **db_rcv_struct** Events.

**To add the InboundEvent Collaboration Rule:**

1 In the components pane of the Schema Designer, select the Collaboration Rules folder.

2 Click the  button to create a new Collaboration Rule.

3 Enter the name **InboundEvent** and click **OK**.

4 Select the newly created Collaboration Rule and click  to display the Collaboration Rule's properties.

5 In the **General** tab, select the **Pass Through** service.

6 Under the Subscriptions tab, select the **FileInEvent** Event Type.

7 Under the Publications tab, select the **db_rcv_in** Event Type.

8 Click **OK** to save and close the Collaboration Rule.

**To add the OutboundEvent Collaboration Rule:**

1 In the components pane of the Schema Designer, select the Collaboration Rules folder.

2 Click the  button to create a new Collaboration Rule.

3 Enter the name **OutboundEvent** and click **OK**.

4 Select the newly created Collaboration Rule and click  to display the Collaboration Rule's properties.

5 In the **General** tab, select the **Pass Through** service.

6 Under the Subscriptions tab, select the **db_rcv_in** Event Type.

7 Under the Publications tab, select the **db_rcv_struct** Event Type.

8 Click **OK** to save and close the Collaboration Rule.

## 4.3.8 Add and Configure the Collaborations

Each of the two e*Ways uses one Collaboration to route the Events through the sample Schema.

- **FileIn_collab** – This collaboration is used by the FileIn e*Way to process the inbound Event and queue it for the **ODBC_rcv** e*Way.
- **ODBC_rcv_collab** – This collaboration subscribes to the Event from the **FileIn_collab** and publishes the Event to the external database.

**To create the FileIn_collab Collaboration:**

1. In the components pane of the Schema Designer, select the **FileIn** e*Way.
2. Click the button to create a new Collaboration.
3. Enter the name **FileIn_collab** and click **OK**.
4. Select the newly created Collaboration and click to display the Collaboration's properties.
5. Select **InboundEvent** from the list of Collaboration Rules.
6. Click **Add** to add a new Subscription.
7. Select the **FileInEvent** Event Type and the **<External>** source.
8. Click **Add** to add a new Publication.
9. Select the **db_rcv_in** Event Type and the **ODBC_IQ** destination.
10. Click **OK** to close the Collaboration's properties.

**To create the ODBC_rcv_collab Collaboration:**

1. In the components pane of the Schema Designer, select the **ODBC_rcv** e*Way.
2. Click the button to create a new Collaboration.
3. Enter the name **ODBC_rcv_collab** and click **OK**.
4. Select the newly created Collaboration and click to display the Collaboration's properties.
5. Select **OutboundEvent** from the list of Collaboration Rules.
6. Click **Add** to add a new Subscription.
7. Select the **db_rcv_in** Event Type and the **FileIn_collab** source.
8. Click **Add** to add a new Publication.
9. Select the **db_rcv_struct** Event Type and the **<External>** destination.
10. Click **OK** to close the Collaboration's properties.

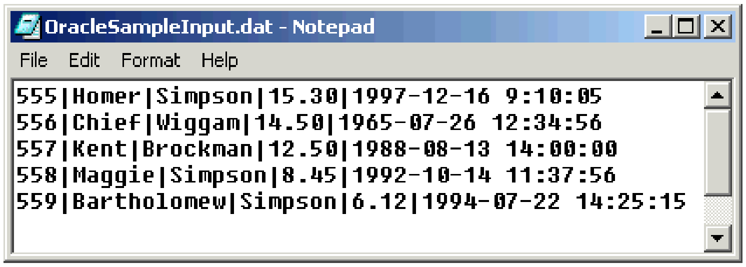
## 4.3.9 Run the Schema

Running the sample Schema requires a sample input file to be created. Once the input file has been created, you can start the Control Broker from a command prompt to execute the Schema. After the Schema has been run, you can use a query utility to query the results in the external database.

### The sample input file

Use a text editor to create an input file to be read by the inbound file e*Way (**FileIn**). The file must be formatted to match the ETD used by the DART script (see **Figure 20 on page 60**). An example of an input file is shown in Figure 23. Save the file to the directory specified in the e*Way's configuration file (such as **c:\egate\data\dart**).

**Figure 23** Sample Input File



```
555|Homer|Simpson|15.30|1997-12-16 9:10:05
556|Chief|Wiggam|14.50|1965-07-26 12:34:56
557|Kent|Brockman|12.50|1988-08-13 14:00:00
558|Maggie|Simpson|8.45|1992-10-14 11:37:56
559|Bartholomew|Simpson|6.12|1994-07-22 14:25:15
```

### To run the Control Broker:

From a command line, type the following command:

```
stccb -ln logical_name -rh registry -rs schema -un user_name -up
password
```

where

*logical_name* is the logical name of the Control Broker,

*registry* is the name of the Registry Host,

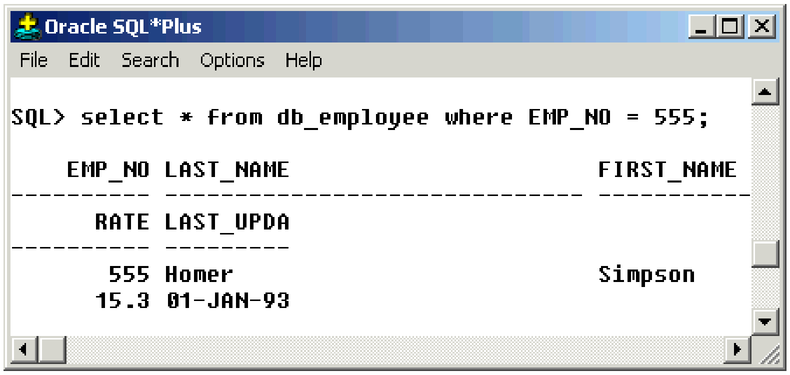
*schema* is the name of the Registry Schema, and

*user_name* and *password* are a valid e*Gate username/password combination.

### To verify the results:

Use an SQL query utility (such as Oracle SQL Plus) to query the results of the output to the Oracle database. Figure 24 shows an example of a query to verify the results of the schema's output based on the input file used by this example.

**Figure 24** Sample Output Console



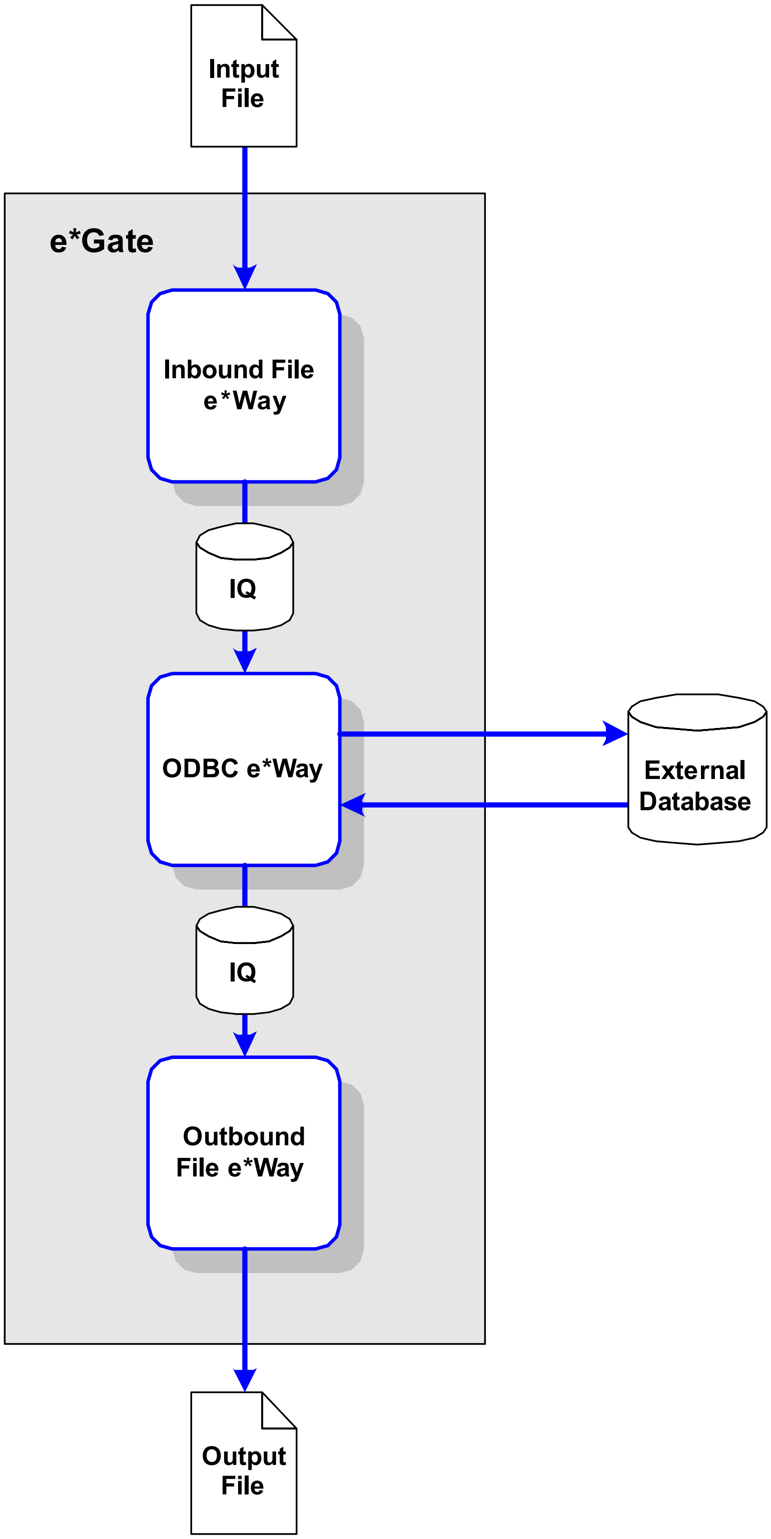
```
SQL> select * from db_employee where EMP_NO = 555;

    EMP_NO LAST_NAME                      FIRST_NAME
---------- ------------------------------ ----------
      RATE LAST_UPDA
---------- ---------
       555 Homer                          Simpson
      15.3 01-JAN-93
```

## 4.4 Sample Two—Polling from an ODBC Database

This section describes how to use the ODBC e*Way in a sample implementation. This sample schema demonstrates the polling of records from an external ODBC database and converting the records into e*Gate Events.

This scenario uses a file e*Way to load an input file containing employee numbers. These employee numbers are converted into e*Gate Events. The ODBC e*Way uses these inbound Events to poll employee records from the external database. As the records are returned to the ODBC e*Way, the Events are published to the outbound IQ. The Outbound file e*Way finally writes the employee records to the output file.

**Figure 25** Polling from an External Database



### Overview of Steps

This sample implementation follows these general steps:

- **“Create the Schema” on page 70**
- **“Create the Event Type Definitions” on page 70**
- **“Add the Event Types” on page 71**
- **“Create the Monk Scripts” on page 72**
- **“Add and Configure the e*Ways” on page 74**
- **“Add the IQs” on page 76**
- **“Create the Collaboration Rules” on page 77**
- **“Add and Configure the Collaborations” on page 77**
- **“Run the Schema” on page 79**

**Note:** *The procedures outlined in this sample are not explained in the same level of detail as in* **Sample One—Publishing e*Gate Events to an ODBC Database** *on page 58. For additional information regarding the configuration of e*Gate components, see* ***Creating an End-to-End Scenario with e*Gate Integrator****.*

## 4.4.1 Create the Schema

The first step in deploying this sample implementation is to create a new Schema.

**To add the new Schema:**

1. Log into the e*Gate Schema Designer.
2. When you are prompted to select a Schema, click **New** to add a new Schema.
3. Name the Schema **ODBC_Sample2**.

## 4.4.2 Create the Event Type Definitions

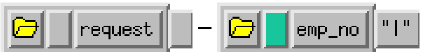
The sample scenario requires two Event Type Definitions. The ETDs are:

- **db_request.ssc** – This ETD is used to format the inbound request Events.
- **db_reply.ssc** – This ETD is used to format the outbound reply Events.

**To create the db_request ETD:**

1. From the e*Gate Schema Designer, click to launch the ETD Editor.
2. Create a new ETD named **db_request.ssc**.
3. Add the nodes and subnodes to create an ETD with the structure shown below:

**Figure 26** The **db_request.ssc** ETD



4 Save the ETD and promote it to Run Time.

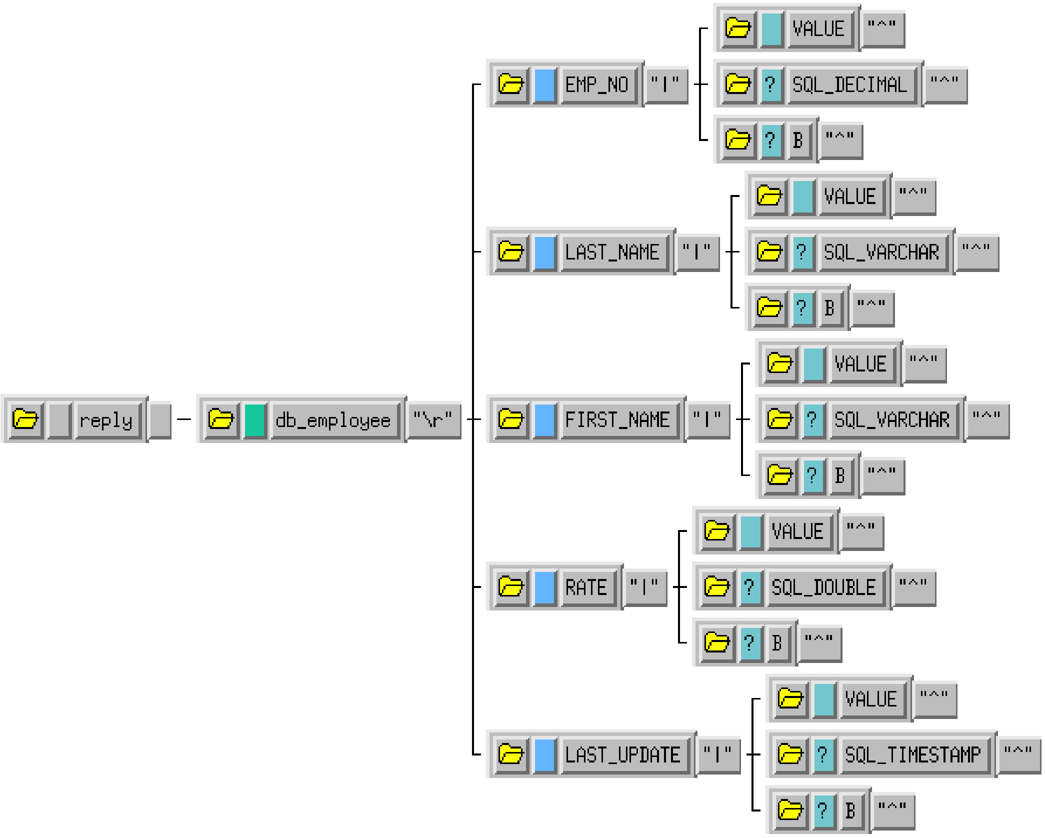
**To create the db_reply ETD:**

1 From the e*Gate Schema Designer, click to launch the ETD Editor.

2 Create a new ETD named **db_reply.ssc**.

3 Add the nodes and subnodes to create an ETD with the structure shown below:

**Figure 27** The **db_reply.ssc** ETD



4 Save the ETD and promote it to Run Time.

## 4.4.3 Add the Event Types

The sample scenario requires six Event Types. The Event Types are:

- **InboundFile** – This Event Type represents the inbound file as it is loaded from the file system.
- **InboundEvent** – This Event Type represents the inbound record that has been converted to an e*Gate Event.
- **PollRequest** – This Event Type represents the request that is sent to the external database.

- **PollReply** – This Event Type represents the reply that is returned by the external database.
- **OutboundEvent** – This Event Type represents the outbound Event to be sent to the external file system.

## 4.4.4 Create the Monk Scripts

This sample implementation uses a DART script (**db_poll.dsc**) to poll the external database.

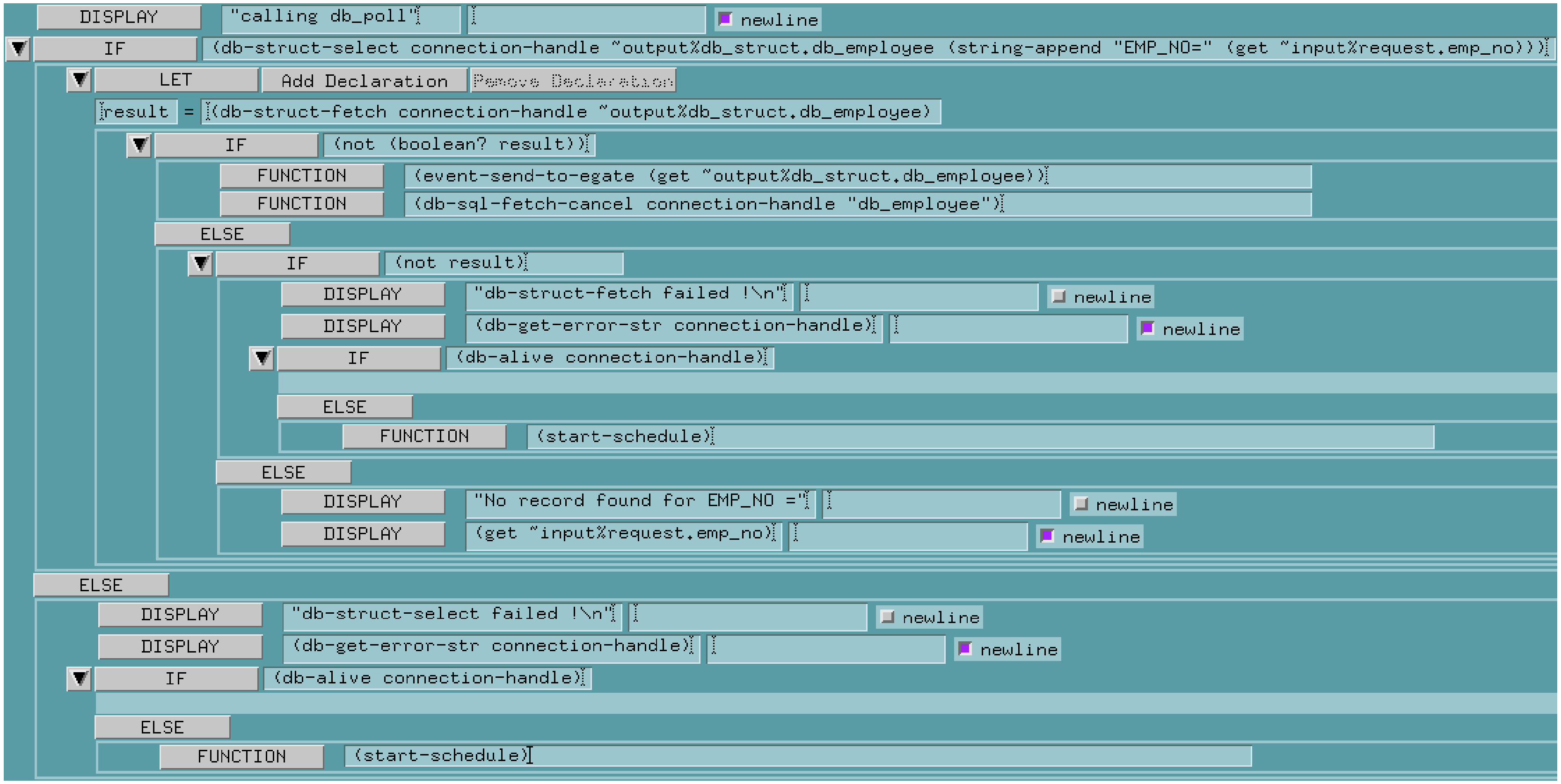
**To create the DART script:**

1. From the e*Gate Schema Designer, click to launch the Collaboration Rules Editor.
2. Click New to create a new DART script.

   The New Collaboration Rules Script dialog will be displayed.
3. Enter the name **db_poll** (with no file extension) as the **File name**.
4. Select **DART Poll** from the list of file types. The extension **.dsc** will be appended to the file name.
5. Click to display the list of source files. Select **db_request.ssc** as the source file.
6. Click to display the list of destination files. Select **db_struct.ssc** as the destination file.
7. Enter the rules as shown in Figure 28.

***Note:*** *The rules shown in Figure 28 use a table named **db_employee**. In order for this sample to work correctly, you must either create a table in your external database called **db_employee** or change each of the references to the table name in your DART script rules as appropriate.*

**Figure 28** The **db_poll.dsc** DART script



```
DISPLAY  "calling db_poll"  ☑ newline
▼ IF  (db-struct-select connection-handle ~output%db_struct.db_employee (string-append "EMP_NO=" (get ~input%request.emp_no)))
    ▼ LET  Add Declaration  Remove Declaration
      result = (db-struct-fetch connection-handle ~output%db_struct.db_employee)
        ▼ IF  (not (boolean? result))
              FUNCTION  (event-send-to-egate (get ~output%db_struct.db_employee))
              FUNCTION  (db-sql-fetch-cancel connection-handle "db_employee")
          ELSE
            ▼ IF  (not result)
                  DISPLAY  "db-struct-fetch failed !\n"  ☐ newline
                  DISPLAY  (db-get-error-str connection-handle)  ☑ newline
                ▼ IF  (db-alive connection-handle)
                  ELSE
                      FUNCTION  (start-schedule)
              ELSE
                  DISPLAY  "No record found for EMP_NO ="  ☐ newline
                  DISPLAY  (get ~input%request.emp_no)  ☑ newline
  ELSE
      DISPLAY  "db-struct-select failed !\n"  ☐ newline
      DISPLAY  (db-get-error-str connection-handle)  ☑ newline
    ▼ IF  (db-alive connection-handle)
      ELSE
          FUNCTION  (start-schedule)
```

8 Click Save to save the script.

9 Close the Collaboration Rules Script Editor.

## 4.4.5 Add and Configure the e*Ways

The sample Schema uses three e*Ways:

- **FileIn** – The **FIleIn** e*Way reads in the input data file and queues it for the ODBC e*Way.
- **ODBC** – The **ODBC** e*Way polls the db_employee table in the external database and queues the returned data for the outbound file e*Way.
- **FileOut** – The **FileOut** e*Way writes the records returned by the ODBC e*Way to the output text file.

### To add and configure the FileIn e*Way:

1 In the components pane of the Schema Designer, select the Control Broker and click to add a new e*Way.

2 Enter **FileIn** for the component name and click **OK**.

3 Select the newly created e*Way and click to display the e*Way's properties.

4 Use the **Find** button to select **stcewfile.exe** as the executable file.

5 Click **New** to create a new configuration file.

6 Enter the parameters for the e*Way as shown in Table 6.

**Table 6** FileIn e*Way Parameters

| Section Name | Parameter | Value |
|---|---|---|
| General Settings | AllowIncoming | Yes |
| | AllowOutgoing | No |
| | Performance Testing | default |
| Outbound (send) settings | All settings | default |
| Poller (inbound) settings | PollDirectory | c:\egate\data\dart |
| | OutputFileName | *.dat |
| | AllOthers | default |
| Performance Testing | All settings | default |

7 Save the e*Way's configuration file and promote it to run time.

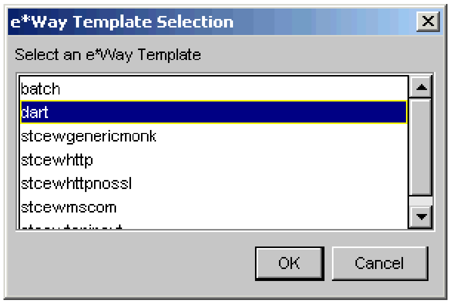
8 In the **Start Up** tab of the e*Way properties, select the **Start automatically** check box.

9 Click **OK** to save the e*Way properties.

### To add and configure the ODBC e*Way:

1. In the components pane of the Schema Designer, select the Control Broker and click to add a new e*Way.
2. Enter **ODBC** for the component name and click **OK**.
3. Select the newly created e*Way and click to display the e*Way's properties.
4. Use the **Find** button to select **stcewgenericmonk.exe** as the executable file.
5. Click **New** to create a new configuration file.
6. Select the **dart** e*Way template and click OK. See Figure 29.

**Figure 29** DART e*Way Template Selection



batch
dart
stcewgenericmonk
stcewhttp
stcewhttpnossl
stcewmscom

7. Enter the parameters for the e*Way as shown in Table 7.

**Table 7** ODBC e*Way Parameters

| Section Name | Parameter | Value |
|---|---|---|
| General Settings | All | default |
| Communication Setup | Start Exchange Data Schedule | Repeatedly, 30 seconds |
| | All others | default |
| Monk Configuration | Process Outgoing Message Function | monk_scripts\common\db_poll.dsc |
| | All others | default |
| Database Setup | Database Type | ODBC |
| | All others | Use local settings |

***Note:*** *Use the appropriate **Database Name**, **User Name**, and **Encrypted Password** according to your local ODBC configuration.*

8. Save the e*Way's configuration file and promote it to run time.
9. In the **Start Up** tab of the e*Way properties, select the **Start automatically** check box.
10. Click **OK** to save the e*Way properties.

**To add and configure the FileIn e*Way:**

1. In the components pane of the Schema Designer, select the Control Broker and click to add a new e*Way.
2. Enter **FileOut** for the component name and click **OK**.
3. Select the newly created e*Way and click to display the e*Way's properties.
4. Use the **Find** button to select **stcewfile.exe** as the executable file.
5. Click **New** to create a new configuration file.
6. Enter the parameters for the e*Way as shown in Table 6.

**Table 8** FileOut e*Way Parameters

| Section Name | Parameter | Value |
|---|---|---|
| General Settings | AllowIncoming | No |
| | AllowOutgoing | Yes |
| | Performance Testing | default |
| Outbound (send) settings | OutputDirectory | c:\egate\data\dart |
| | OutputFileName | PollOutput%d.dat |
| | All Others | default |
| Poller (inbound) settings | All | default |
| Performance Testing | All | default |

7. Save the e*Way's configuration file and promote it to run time.
8. In the **Start Up** tab of the e*Way properties, select the **Start automatically** check box.
9. Click **OK** to save the e*Way properties.

## 4.4.6 Add the IQs

This sample Schema requires two Intelligent Queues: ODBC_1IQ and ODBC_2IQ.

**To add the IQs:**

1. In the components pane of the Schema Designer, select the IQ manager. Click to create the first new IQ.
2. Enter the name **ODBC_1IQ** and click **Apply** to save the first IQ.
3. Enter the name **ODBC_2IQ** and click **OK** to save the second IQ.
4. Select the IQ Manager and click to display the IQ Manager's properties.
5. In the **Start Up** tab of the IQ Manager's properties, select the **Start automatically** check box.

6 Click **OK** to save the IQ Manager's properties.

## 4.4.7 Create the Collaboration Rules

This sample schema uses four Collaboration Rules:

- **FileIn** – This Collaboration Rule is used by the **FileIn** e*Way's Collaboration to transform the **InboundFile** Events into **InboundEvent** Events.
- **ODBCRequest** – This Collaboration Rule is used by the **ODBC** e*Way's Collaboration to transform the **InboundEvent** Events into **PollRequest** Events.
- **ODBCReply** – This Collaboration Rule is used by the **ODBC** e*Way's Collaboration to transform the **PollRequest** Events into **PollReply** Events.
- **FileOut** – This Collaboration Rule is used by the **FileOut** e*Way's Collaboration to transform the **PollReply** Events into **OutboundEvent** Events.

**To add the FileIn Collaboration Rule:**

1 In the components pane of the Schema Designer, select the Collaboration Rules folder.

2 Click the button to create a new Collaboration Rule.

3 Enter the name **FileIn** and click **OK**.

4 Select the newly created Collaboration Rule and click to display the Collaboration Rule's properties.

5 In the **General** tab, select the **Pass Through** service.

6 Under the Subscriptions tab, select the **InboundFile** Event Type.

7 Under the Publications tab, select the **InboundEvent** Event Type.

8 Click **OK** to save and close the Collaboration Rule.

**To add the remaining Collaboration Rules:**

Follow the same steps used to add the **FileIn** Collaboration Rule using the names and Event Types shown at the beginning of this section.

## 4.4.8 Add and Configure the Collaborations

This sample schema uses four Collaborations:

- **FileIn_collab** – This Collaboration is used to transform the **InboundFile** Events into **InboundEvent** Events.
- **ODBCRequest_collab** – This Collaboration is used to transform the **InboundEvent** Events into **PollRequest** Events.
- **ODBCReply_collab** – This Collaboration is used to transform the **PollRequest** Events into **PollReply** Events.
- **FileOut_collab** – This Collaboration is used to transform the **PollReply** Events into **OutboundEvent** Events.

### To create the FileIn_collab Collaboration:

1. In the components pane of the Schema Designer, select the **FileIn** e*Way.
2. Click the button to create a new Collaboration.
3. Enter the name **FileIn_collab** and click **OK**.
4. Select the newly created Collaboration and click to display the Collaboration's properties.
5. Select **InboundFile** from the list of Collaboration Rules.
6. Click **Add** to add a new Subscription.
7. Select the **InboundEvent** Event Type and the **<External>** source.
8. Click **Add** to add a new Publication.
9. Select the **InboundEvent** Event Type and the **ODBC_1IQ** destination.
10. Click **OK** to close the Collaboration's properties.

### To create the ODBCRequest_collab Collaboration:

1. In the components pane of the Schema Designer, select the **ODBC** e*Way.
2. Click the button to create a new Collaboration.
3. Enter the name **ODBCRequest_collab** and click **OK**.
4. Select the newly created Collaboration and click to display the Collaboration's properties.
5. Select **ODBCRequest** from the list of Collaboration Rules.
6. Click **Add** to add a new Subscription.
7. Select the **InboundFile** Event Type and the **FileIn_Collab** source.
8. Click **Add** to add a new Publication.
9. Select the **ODBCRequest** Event Type and the **<External>** destination.
10. Click **OK** to close the Collaboration's properties.

### To create the ODBCReply_collab Collaboration:

1. In the components pane of the Schema Designer, select the **ODBC** e*Way.
2. Click the button to create a new Collaboration.
3. Enter the name **ODBCReply_collab** and click **OK**.
4. Select the newly created Collaboration and click to display the Collaboration's properties.
5. Select **ODBCReply** from the list of Collaboration Rules.
6. Click **Add** to add a new Subscription.
7. Select the **ODBCRequest** Event Type and the **<External>** source.

8. Click **Add** to add a new Publication.
9. Select the **ODBCReply** Event Type and the **ODBC_2IQ** destination.
10. Click **OK** to close the Collaboration's properties.

**To create the FileOut_collab Collaboration:**

1. In the components pane of the Schema Designer, select the **FileOut** e*Way.
2. Click the button to create a new Collaboration.
3. Enter the name **FileOut_collab** and click **OK**.
4. Select the newly created Collaboration and click to display the Collaboration's properties.
5. Select **FileOut** from the list of Collaboration Rules.
6. Click **Add** to add a new Subscription.
7. Select the **ODBCReply** Event Type and the **ODBCReply_collab** source.
8. Click **Add** to add a new Publication.
9. Select the **OutboundEvent** Event Type and the **<External>** destination.
10. Click **OK** to close the Collaboration's properties.

## 4.4.9 Run the Schema

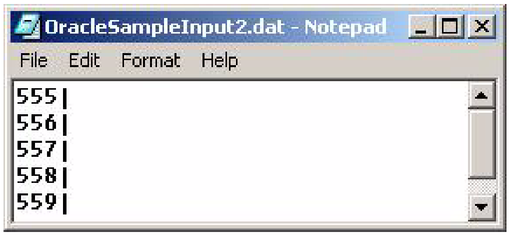
Running the sample Schema requires a sample input file to be created. Once the input file has been created, you can start the Control Broker from a command prompt to execute the Schema. After the Schema has been run, you can view the results in the output file.

### The sample input file

Use a text editor to create an input file to be ready by the inbound file e*Way (**FileIn**). The file must be formatted to match the simple ETD used by the DART script (see Figure 26 on page 70). An example of an input file is shown in Figure 30. Save the file to the directory specified in the e*Way's configuration file (such as **c:\egate\data\dart**).

***Note:*** *The "employee numbers" used in this example must exist in your external database. The sample shown below uses employee numbers that exist from the records in the previous sample schema.*

**Figure 30** Sample Input File



```
555|
556|
557|
558|
559|
```

### To run the Control Broker:

From a command line, type the following command:

```
stccb -ln logical_name -rh registry -rs ODBC_Sample2 -un user_name
-up password
```

where

*logical_name* is the logical name of the Control Broker,

*registry* is the name of the Registry Host, and

*user_name* and *password* are a valid e*Gate username/password combination.

### To verify the results:

Use a text editor to view the records that were written to the output file specified by the FileOut e*Way. The records should correspond to the records in the external database.

Chapter 5

# ODBC e*Way Functions

The functions described in this chapter control the ODBC e*Way's basic operations as well as those needed for database access.

**Note:** *The functions described in this section can only be used by the functions defined within the e*Way's configuration file. None of the functions are available to Collaboration Rules scripts executed by the e*Way.*

**This Chapter Explains:**

## 5.1 Basic Functions

The functions in this category control the e*Way's most basic operations.

The basic functions are:

## event-send-to-egate

### Syntax

```
(event-send-to-egate string)
```

### Description

**event-send-to-egate** sends an Event from the e*Way. If the external collaboration(s) is successful in publishing the Event to the outbound queue, the function will return **#t**, otherwise **#f**.

### Parameters

| Name | Type | Description |
|---|---|---|
| string | string | The data to be sent to the e*Gate system |

### Return Values

**Boolean**
Returns **#t** when successful and **#f** when an error occurs.

### Throws

None.

### Additional information

This function can be called by any e*Way function when it is necessary to send data to the e*Gate system in a blocking fashion.

# get-logical-name

## Syntax

```
(get-logical-name)
```

## Description

**get-logical-name** returns the logical name of the e*Way.

## Parameters

None.

## Return Values

**string**
Returns the name of the e*Way (as defined by the Schema Designer).

## Throws

None.

## send-external-down

### Syntax

```
(send-external-down)
```

### Description

**send-external down** instructs the e*Way that the connection to the external system is down.

### Parameters

None.

### Return Values

None.

### Throws

None.

## send-external-up

### Syntax

```
(send-external-up)
```

### Description

**send-external-up** instructs the e*Way that the connection to the external system is up.

### Parameters

None.

### Return Values

None.

### Throws

None.

## shutdown-request

### Syntax

```
(shutdown-request)
```

### Description

**shutdown-request** completes the e*Gate shutdown procedure that was initiated by the Control Broker but was interrupted by returning a non-null value within the **Shutdown Command Notification Function** (see **"Shutdown Command Notification Function" on page 45**). Once this function is called, shutdown proceeds immediately.

Once interrupted, the e*Way's shutdown cannot proceed until this Monk function is called. If you do interrupt an e*Way shutdown, we recommend that you complete the process in a timely fashion.

### Parameters

None.

### Return Values

None.

### Throws

None.

## start-schedule

### Syntax

```
(start-schedule)
```

### Description

**start-schedule** requests that the e*Way execute the **Exchange Data with External Function** specified within the e*Way's configuration file. Does not effect any defined schedules.

### Parameters

None.

### Return Values

None.

### Throws

None.

## stop-schedule

### Syntax

```
(stop-schedule)
```

### Description

**stop-schedule** requests that the e*Way halt execution of the **Exchange Data with External Function** specified within the e*Way's configuration file. Execution will be stopped when the e*Way concludes any open transaction. Does not effect any defined schedules, and does not halt the e*Way process itself.

### Parameters

None.

### Return Values

None.

### Throws

None.

## 5.2 Standard e*Way Functions

The functions in this category control the e*Way's standard operations.

The standard functions are:

**db-stdver-conn-estab** on page 90

**db-stdver-conn-shutdown** on page 92

**db-stdver-conn-ver** on page 93

**db-stdver-data-exchg** on page 95

**db-stdver-data-exchg-stub** on page 96

**db-stdver-init** on page 97

**db-stdver-neg-ack** on page 98

**db-stdver-pos-ack** on page 99

**db-stdver-proc-outgoing** on page 100

**db-stdver-proc-outgoing-stub** on page 102

**db-stdver-shutdown** on page 104

**db-stdver-startup** on page 105

## db-stdver-conn-estab

### Syntax

```
(db-stdver-conn-estab)
```

### Description

**db-stdver-conn-estab** is used to establish external system connection.The following tasks are performed by this function:

- construct a new connection handle
- call db-long to connect to database
- setup timestamp format if required
- setup maximum long data buffer limit if required
- bind dynamic SQL statement and stored procedures.

### Parameters

None.

### Return Values

### A string

**UP** or **SUCCESS** if connection established, anything else if connection not established.

### Throws

None.

### Additional Information

In order to use the standard database time format, the following function call has been added to this function (immediately before the call to the **db-bind** function):

```
(db-std-timestamp-format connection-handle)
```

To override the use of the standard database time format, the **db-std-timestamp-format** function call should be removed.

For "Maximum Long Data Size" the ODBC library allocates an internal buffer for each SQL_LONGVARCHAR and SQL_LONGVARBINARY data, when the SQL statement or stored procedure that contains these data types are bound. The default size of each internal data buffer is 1024K(1048576) bytes. If the user needs to handle long data larger than this default value, add the following function call to specify the maximum data size:

```
(db-max-long-data-size connection-handle maximum-data-size)
```

See **db-max-long-data-size** on page 116 for more information.

### Examples

```
(define db-stdver-conn-estab
 (lambda (  )
  (let ((result "DOWN")(last_dberr ""))
  (display "[++] Executing e*Way external connection establishment
function.")
 (display "db-stdver-conn-estab: logging into the database with:\n")
 (display "DATABASE NAME = ")
 (display DATABASE_SETUP_DATABASE_NAME)
 (newline )
 (display "USER NAME = ")
 (display DATABASE_SETUP_USER_NAME)
 (newline )
 (set! connection-handle (make-connection-handle))
 (if (connection-handle? connection-handle)
  (begin
   (if (db-login connection-handle DATABASE_SETUP_DATABASE_NAME
DATABASE_SETUP_USER_NAME DATABASE_SETUP_ENCRYPTED_PASSWORD)
    (begin
          (db-bind )
     (set! result "UP")
    )
    (begin
     (set! last_dberr (db-get-error-str connection-handle))
     (display last_dberr)
     (event-send "ALERTCAT_OPERATIONAL" "ALERTSUBCAT_CANTCONN"
"ALERTINFO_FATAL" "0" "Cannot connect to database" (string-append
"Failed to connect to database: " DATABASE_SETUP_DATABASE_NAME "with
error" last_dberr) 0 (list))
     (newline )
     (db-logout connection-handle)
     (set! result "DOWN")
    )
   )
  )
  (begin
   (set! result "DOWN")
   (display "Failed to create connection handle.")
   (event-send "ALERTCAT_OPERATIONAL" "ALERTSUBCAT_UNUSABLE"
"ALERTINFO_FATAL" "0" "database connection handle creation error"
"Failed to create database connection handle" 0 (list))
  )
 )
 result
  )
))
```

## db-stdver-conn-shutdown

### Syntax

```
(db-stdver-conn-shutdown string)
```

### Description

**db-stdver-conn-shutdown** is called by the system to request that the interface disconnect from the external system, preparing for a suspend/reload cycle. Any return value indicates that the suspend can occur immediately, and the interface will be placed in the down state.

### Parameters

| Name | Type | Description |
|---|---|---|
| string | string | When the e*Way calls this function, it will pass the string "SUSPEND_NOTIFICATION" as the parameter. |

### Return Values

**A string**

A return of "SUCCESS" indicates that the external is ready to suspend.

### Throws

None.

### Examples

```
(define db-stdver-conn-shutdown
 (lambda ( message-string )
  (let ((result "SUCCESS"))
 (comment "Std e*Way connection shutdown function" "[++] Usage:
Function called by system to request that the interface disconnect
from the external system, preparing for a suspend/reload cycle. Any
return value indicates that the suspend can occur immediately, and the
interface will be placed in the down state. [++] Input to expect:
Function should not expect input. [++] Expected return values:
anything indicates that the external is ready to suspend.n")
 (comment "db-stdver-conn-shutdown [++] Implementation specific
comment" "none")
 (display "[++] Executing e*Way external connection shutdown
function.")
 (display message-string)
 (db-logout connection-handle)
 result
  )
))
```

## db-stdver-conn-ver

### Syntax

```
(db-stdver-conn-ver)
```

### Description

**db-stdver-conn-ver** is used to verify whether the external system connection is established.

### Parameters

None.

### Return Values

### A string

**UP** or **SUCCESS** if connection established, anything else if connection not established.

### Throws

None.

### Additional Information

To use standard database time format, add the following function call to this function: (**db-std-timestamp-format** *connection-handle*) after the (**db-bind**) call.

This SQL statement is designed for DBMSs other than Oracle; the use of this function occasionally results in an error in the e*Way's log file. Despite the error, the function will complete successfully.

***Note:*** *To users of earlier versions of DART:* ***db-check-connect*** *calls should be replaced with* ***db-alive*** *calls.*

### Examples

```
(define db-stdver-conn-ver
 (lambda (  )
  (let ((result "DOWN")(last_dberr ""))
   (display "[++] Executing e*Way external connection verification
function.")
 (display "db-stdver-conn-ver: checking connection status...\n")
 (cond ((string=? STCDB "SYBASE") (db-sql-select connection-handle
"verify" "select getdate()")) ((string=? STCDB "ORACLE8i") (db-sql-
select connection-handle "verify" "select sysdate from dual"))
((string=? STCDB "ORACLE8") (db-sql-select connection-handle "verify"
"select sysdate from dual")) ((string=? STCDB "ORACLE7") (db-sql-
select connection-handle "verify" "select sysdate from dual")) (else
(db-sql-select connection-handle "verify" "select {fn NOW()}")))
 (if (db-alive connection-handle)
  (begin
   (db-sql-fetch-cancel connection-handle "verify")
   (set! result "UP")
  )
  (begin
   (set! last_dberr (db-get-error-str connection-handle))
   (display last_dberr)
   (event-send "ALERTCAT_OPERATIONAL" "ALERTSUBCAT_LOSTCONN"
"ALERTINFO_FATAL" "0" "Lost connection to database" (string-append
"Lost connection to database: " DATABASE_SETUP_DATABASE_NAME "with
error" last_dberr) 0 (list))
   (set! result "DOWN")
  )
 )
 result
  )
))
```

## db-stdver-data-exchg

### Syntax

```
(db-stdver-data-exchg)
```

### Description

**db-stdver-data-exchg** is used for sending a received Event from the external system to e*Gate. The function expects no input.

### Parameters

None.

### Return Values

### A string

A message-string indicates a successful operation. The Event is sent to e*Gate

An empty string indicates a successful operation. Nothing is sent to e*Gate.

CONNERR indicates the loss of connection with the external, client moves to a down state and attempts to connect. Upon reconnecting, this function will be re-executed with the same input message.

### Throws

None.

### Examples

```
(define db-stdver-data-exchg
 (lambda (  )
  (let ((result ""))
  (display "[++] Executing e*Way external data exchange function.")
 result
  )
))
```

## db-stdver-data-exchg-stub

### Syntax

```
(db-stdver-data-exchg-stub)
```

### Description

**db-stdver-data-exchg-stub** is used as a place holder for the function entry point for sending an Event from the external system to e*Gate. When the interface is configured as an outbound only connection, this function should not be called. The function expects no input.

### Parameters

None.

### Return Values

### A string

A message-string indicates a successful operation. The Event is sent to e*Gate

An empty string indicates a successful operation. Nothing is sent to e*Gate.

CONNERR indicates the loss of connection with the external, client moves to a down state and attempts to connect. Upon reconnecting, this function will be re-executed with the same input message.

### Throws

None.

### Examples

```
(define db-stdver-data-exchg-stub
 (lambda (  )
  (let ((result ""))
  (event-send "ALERTCAT_OPERATIONAL" "ALERTSUBCAT_INTEREST"
"ALERTINFO_NONE" "0" "Possible configuration error." "Default eway
data exchange function called." 0 (list))
 result
  )
))
```

## db-stdver-init

### Syntax

```
(db-stdver-init)
```

### Description

**db-stdver-init** begins the initialization process for the e*Way. The function loads all of the monk extension library files that the other e*Way functions will access.

### Parameters

None.

### Return Values

### A string

If a **FAILURE** string is returned, the e*Way will shutdown. Any other return indicates success.

### Throws

None.

# db-stdver-neg-ack

## Syntax

```
(db-stdver-neg-ack message-string)
```

## Description

**db-stdver-neg-ack** is used to send a negative acknowledgement to the external system, and for post processing after failing to send data to e*Gate.

## Parameters

| Name | Description |
|---|---|
| message-string | The Event for which a negative acknowledgment is sent. |

## Return Values

**A string**

An empty string indicates a successful operation.

**CONNERR** indicates a loss of connection with the external, client moves to a down state and attempts to connect, on reconnect neg-ack function will be re-executed.

## Throws

None.

## Examples

```
(define db-stdver-neg-ack
 (lambda ( message-string )
  (let ((result ""))
 (  (display "[++] Executing e*Way external negative acknowledgment
function.")
 (display message-string)
 result
  )
))
```

# db-stdver-pos-ack

## Syntax

```
(db-stdver-pos-ack message-string)
```

## Description

**db-stdver-pos-ack** is used to send a positive acknowledgement to the external system, and for post processing after successfully sending data to e*Gate.

## Parameters

| Name | Description |
|---|---|
| message-string | The Event for which an acknowledgment is sent. |

## Return Values

### A string

An empty string indicates a successful operation. The e*Way will then be able to proceed with the next request.

**CONNERR** indicates a loss of connection with the external, client moves to a down state and attempts to connect, on reconnect pos-ack function will be re-executed.

## Throws

None.

## Examples

```
(define db-stdver-pos-ack
 (lambda ( message-string )
  (let ((result ""))
  (display "[++] Executing e*Way external positive acknowledgement
function.")
 (display message-string)
 result
  )
))
```

## db-stdver-proc-outgoing

### Syntax

```
(db-stdver-proc-outgoing message-string)
```

### Description

**db-stdver-proc-outgoing** is used for sending a received message (Event) from e*Gate to the external system.

### Parameters

| Name | Type | Description |
|---|---|---|
| message-string | string | The Event to be processed. |

### Return Values

**A string**

An empty string indicates a successful operation.

**RESEND** causes the message to be immediately resent. The e*Way will compare the number of attempts it has made to send the Event to the number specified in the Max Resends per Messages parameter, and does one of the following:

1. If the number of attempts does not exceed the maximum, the e*Way will pause the number of seconds specified by the **Resend Timeout** parameter, increment the "resend attempts" counter for that message, then repeat the attempt to send the message.
2. If the number of attempts exceeds the maximum, the function returns false and rolls back the message to the e*Gate IQ from which it was obtained.

**CONNERR** indicates that there is a problem communicating with the external system. First, the e*Way will pause the number of seconds specified by the **Resend Timeout** parameter. Then, the e*Way will call the **External Connection Establishment function according to the Down Timeout schedule, and** will roll back the message (Event) to the IQ from which it was obtained.

**DATAERR** indicates that there is a problem with the message (Event) data itself. First, the e*Way will pause the number of seconds specified by the **Resend Timeout** parameter. Then, the e*Way increments its "failed message (Event)" counter, and rolls back the message (Event) to the IQ from which it was obtained. If the e*Way's journal is enabled (see **Journal File Name** on page 26) the message (Event) will be journaled.

If a string other than the following is returned, the e*Way will create an entry in the log file indicating that an attempt has been made to access an unsupported function.

### Throws

None.

### Examples

```
(define db-stdver-proc-outgoing
 (lambda ( message-string )
  (let ((result ""))
  (display "[++] Executing e*Way external process outgoing message
function.")
 (display message-string)
 result
  )
))
```

## db-stdver-proc-outgoing-stub

### Syntax

```
(db-stdver-proc-outgoing-stub message-string)
```

### Description

**db-stdver-proc-outgoing-stub** is used as a place holder for the function entry point for sending an Event received from e*Gate to the external system. When the interface is configured as an inbound only connection, this function should not be used. This function is used to catch configuration problems.

### Parameters

| Name | Type | Description |
|---|---|---|
| message-string | string | The Event to be processed. |

### Return Values

**A string**

An empty string indicates a successful operation.

**RESEND** causes the message to be immediately resent. The e*Way will compare the number of attempts it has made to send the Event to the number specified in the Max Resends per Messages parameter, and does one of the following:

1. If the number of attempts does not exceed the maximum, the e*Way will pause the number of seconds specified by the **Resend Timeout** parameter, increment the “resend attempts” counter for that message, then repeat the attempt to send the message.
2. If the number of attempts exceeds the maximum, the function returns false and rolls back the message to the e*Gate IQ from which it was obtained.

**CONNERR** indicates that there is a problem communicating with the external system. First, the e*Way will pause the number of seconds specified by the **Resend Timeout** parameter. Then, the e*Way will call the **External Connection Establishment function according to the Down Timeout schedule, and** will roll back the message (Event) to the IQ from which it was obtained.

**DATAERR** indicates that there is a problem with the message (Event) data itself. First, the e*Way will pause the number of seconds specified by the **Resend Timeout** parameter. Then, the e*Way increments its “failed message (Event)” counter, and rolls back the message (Event) to the IQ from which it was obtained. If the e*Way’s journal is enabled (see **Journal File Name** on page 26) the message (Event) will be journaled.

If a string other than the following is returned, the e*Way will create an entry in the log file indicating that an attempt has been made to access an unsupported function.

### Throws

None.

### Examples

```
(define db-stdver-proc-outgoing-stub
 (lambda ( message-string )
  (let ((result ""))
  (display "[++] Executing e*Way external process outgoing message
function stub.")
 (display message-string)
 (event-send "ALERTCAT_OPERATIONAL" "ALERTSUBCAT_INTEREST"
"ALERTINFO_NONE" "0" "Possible configuration error." (string-append
"Default eway process outgoing msg function passed following message:
" msg) 0 (list))
 result
  )
))
```

## db-stdver-shutdown

### Syntax

```
(db-stdver-shutdown shutdown_notification)
```

### Description

**db-stdver-shutdown** is called by the system to request that the external shutdown, a return value of SUCCESS indicates that the shutdown can occur immediately, any other return value indicates that the shutdown Event must be delayed. The user is then required to execute a shutdown-request call from within a monk function to allow the requested shutdown process to continue.

### Parameters

| Name | Type | Description |
|---|---|---|
| shutdown_notification | string | When the e*Way calls this function, it will pass the string "SHUTDOWN_NOTIFICATION" as the parameter. |

### Return Values

**A string**

**SUCCESS** allows an immediate shutdown to occur, anything else delays shutdown until (shutdown-request) is executed successfully.

### Throws

None.

### Examples

```
(define db-stdver-shutdown
 (lambda ( message-string )
  (let ((result "SUCCESS"))
  (display "[++] Executing e*Way external shutdown command
notification function.")
 result
  )
))
```

## db-stdver-startup

### Syntax

```
(db-stdver-startup)
```

### Description

**db-stdver-startup** is used for instance specific function loads and invokes setup.

### Parameters

None.

### Return Values

**A string**

**FAILURE** causes shutdown of the e*Way. Any other return indicates success.

### Throws

None.

### Examples

```
(define db-stdver-startup
 (lambda (   )
  (let ((result "SUCCESS"))
  (display "[++] Executing e*Way external startup function.")
 result
  )
))
```

## 5.3 General Connection Functions

The functions in this category control the e*Way's database connection operations.

The general connection functions are:

**connection-handle?** on page 107

**db-alive** on page 108

**db-commit** on page 110

**db-get-error-str** on page 111

**db-login** on page 113

**db-logout** on page 115

**db-max-long-data-size** on page 116

**db-rollback** on page 117

**make-connection-handle** on page 118

**statement-handle?** on page 119

## connection-handle?

### Syntax

```
(connection-handle? any-variable)
```

### Description

**connection-handle**? determines whether or not the input argument is a *connection handle* datatype.

### Parameters

This function requires a single variable of any datatype.

### Return Values

### Boolean

Returns **#t** (true) if the argument is a connection handle; otherwise, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

### Throws

None.

### Examples

```
(define hdbc (make-connection-handle))
(if (not (connection-handle? hdbc))
    (display (db-get-error-str hdbc))
)
```

### Explanation

The above example creates a connection handle called hdbc. An error message is displayed if the newly defined hdbc is not a connection handle.

## db-alive

### Syntax

```
(db-alive connection-handle)
```

### Description

**db-alive** is used to determine if the cause of a failing ODBC operation is due to a broken connection. It returns whether or not the database connection was alive during the last call to any ODBC procedure that sends commands to the database server.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |

### Return Values

**Boolean**

Returns **#t** (true) if the connection to the database server is still alive; otherwise, returns **#f** (false) if the connection to the database server is either dead or down. Use **db-get-error-str** to retrieve the error message.

### Throws

None.

### Examples

```
(if (db-login hdbc “dsn” “uid” “pd”)
    (begin
        (define sql_statement “select * from person where sex = ‘M’”)
        (do ((status #t)) ((not status))
            (if (db-sql-select hdbc “male” sql_statement)
                (begin
                    ...
                )
                (begin
                    (display (db-get-error-str hdbc))
                    (set! status (db-alive hdbc))
                )
            )
        )
        (display “lost database connection !\n”))
        (db-logout hdbc))
    )
)
```

### Explanation

The example above illustrates an application that is looking for a certain record in the person table of the “Payroll” database. The function will exit the loop only if it loses the connection to the database.

### Notes

1 Most ODBC procedures can detect a dead connection handle except **db-commit** and **db-rollback**. Therefore, when the ODBC procedure returns false, users must check for loss of connection.

2 Once the **db-alive** returns **#f** to indicate either a dead connection handle or an unavailable database server, all the subsequent ODBC function calls associated with that connection handle will not be executed, with the exception of **db-logout**. Each of these procedures will return false with a “lost database connection” error message.

3 Once the ODBC e*Way determines the connection handle is not alive, the only course of action the user can take is to log out from that connection handle, redefine a new connection handle, and try to reconnect to the database.

## db-commit

### Syntax

```
(db-commit connection-handle)
```

### Description

**db-commit** performs all transactions specified by the connection.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |

### Return Values

**Boolean**

Returns **#t** (true) if successful; otherwise, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

### Throws

None.

### Examples

```
...
(if
    (and
        (db-sql-execute hdbc "delete from employee where first_name =
'John'")
        (db-sql-execute hdbc "update employee set first_name = 'Mary'
where ssn = 123456789")
    )
    (db-commit hdbc)
    (begin
        (display (db-get-error-str hdbc))
        (db-rollback hdbc)
    )
)
...
```

### Explanation

This example shows that if the application can successfully delete "John's record" and update "Mary's record" it will commit the transaction specified by the connection. Otherwise, it prints out the error message and rolls back the transaction.

## db-get-error-str

### Syntax

```
(db-get-error-str connection-handle)
```

### Description

**db-get-error-str** returns the last error message, and is used when the function returns a **#f** value.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |

### Return Values

**A string**

A simple error message is returned.

To parse the return error message when it contains an error, use the two standard files that define the error message structure and display the contents of each component of the error message.

```
ODBC - odbcmsg.ssc, odbcmsg_display.monk
```

### Throws

None.

### Examples

**Scenario #1 — sample code for db-get-error-str**

```
...
(if (db-sql-execute hdbc "delete from employee" where
first_name='John')
    (db-commit hdbc)
    (display (db-get-error-str hdbc))
)
...
```

### Explanation

This example shows that if the application can successfully delete “John’s record” it will commit the transaction. Otherwise, the application will print out the error message and roll back the same transaction. Each commit begins a new transaction automatically.

```
(if (db-login hdbc dsn uid pwd)
   (begin
      (display "database login succeed !\n")
      (if (db-sql-execute hdbc "INSERT INTO UNKNOWN VALUES (NULL)")
         (db-commit hdbc)
         (odbcmsg-display (db-get-error-str hdbc))
      )
      (if (not (db-logout hdbc))
         (odbcmsg-display (db-get-error-str hdbc))
      )
```

```
        )
        (odbcmsg-display (db-get-error-str hdbc))
    )
```

### Program output of the above example:

### Output of (db-get-error-str hdbc)

```
ODBC|S0002|942|INTERSOLV|ODBC Oracle driver|Oracle|ORA-00942: table
or view does not exist
DART|63|STCDB_X_conn_sql_exec_len||unable to execute SQL statement
```

### Output of (odbcmsg-display (db-get-error-str hdbc))

```
ODBC message #0:
msg_source      : ODBC
sql_state       : S0002
native_code     : 942
drv_vendor      : INTERSOLV
component       : ODBC Oracle driver
err_source      : Oracle
msg_string      : ORA-00942: table or view does not exist

DART message #0:
msg_source      : DART
msg_number      : 63
function        : STCDB_X_conn_sql_exec_len
err_item        :
msg_string      : unable to execute SQL statement
```

# db-login

## Syntax

```
(db-login  connection-handle  data-source  user-name  password)
```

## Description

**db-login** allocates the resources and performs login to a database system.

This function requires an encrypted password. If you have specified a password in the Database Setup section of the e*Way Editor, it has already been encrypted. (See **“Database Setup” on page 45**.)

If you define the password within a monk function (which is not encrypted), you must use the monk function **encrypt-password** found in the e*Gate Monk extension library stc_monkext.dll:

```
encrypt-password  encryption key   plain password
```

where encryption key is public knowledge, i.e., in this case user id, and plain password is the password to be encrypted.

The standard encrypt-password function returns an encrypted password string to be used with db-login.

## Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| data-source | string | The name of the data source. |
| user-name | string | The database user login name. |
| password | string | The database user login password. |

***Note:*** *The data_source, user_name, and password must not be an empty string.*

## Return Values

**Boolean**

Returns **#t** (true) if the argument is a connection handle; otherwise, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

## Throws

None.

## Examples

```
...
(define hdbc (make-connection-handle) )
(define uid "James")
(define pwd (encrypt-password uid "12345"))
(if (db-login hdbc ”Payroll” “James” “12345”)
    ...
)
```

### Explanation

The above example shows how to use the connection handle (hdbc) to log into the data source “Payroll” as “James” with the password “12345.”

## db-logout

### Syntax

```
(db-logout  connection-handle)
```

### Description

**db-logout** performs a disconnect from the database system and releases the connection handle resources.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |

### Return Values

**Boolean**

Returns **#t** (true) if successful; otherwise, returns **#f** (false).Use **db-get-error-str** to retrieve the error message.

### Throws

None.

### Examples

```
...
(define hdbc (make-connection handle) )
(define uid "James")
(define pwd (encrypt-password uid "12345"))
(if (db-login hdbc “Payroll” “James” “12345”)
    ...
    (db-logout hdbc)
)
...
```

### Explanation

The above example shows how to disconnect from a database. For every **db-login**, there should be a corresponding **db-logout**.

### Notes

Make sure you roll back or commit a transaction before you call **db-logout**. If a transaction is neither committed nor rolled back, it will be automatically rolled back before logout.

## db-max-long-data-size

### Syntax

```
(db-max-long-data-size connection-handle size)
```

### Description

**db-max-long-data-size** specifies the maximum buffer size for the long data.(SQL_LONGVARCHAR, SQL_LONGVARBINARY) Long data may have a range in size up to 2 gigabytes ($2x10^9$). In order to limit the memory consumption of the ODBC library, it is necessary to use this function to specify the maximum data size expected. Long data larger than the specified size will be truncated. This data size will be used for buffer allocation for both long data columns as well as long data parameters.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| size | integer | This parameter is used to identify the buffer size of the specified long data type. Note: The default buffer size is 1 megabyte. |

### Return Values

**Boolean**

Returns #t (true) if successful; and If unsuccessful, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

### Throws

None.

### Additional Information

The default maximum buffer size for long data type is 1 megabyte (1048576). It is not necessary to call this function unless the long data is in excess of 1 megabyte.

# db-rollback

## Syntax

```
(db-rollback connection-handle)
```

## Description

**db-rollback** rolls back the entire transaction for the connection.

## Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |

## Return Values

**Boolean**

Returns **#t** (true) if successful; otherwise, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

## Throws

None.

## Examples

```
...
(if
    (and
        (db-sql-execute hdbc “delete from employee where first_name =
‘John’”)
        (db-sql-execute hdbc “update employee set first_name = ‘Mary’
where ssn = 123456789”)
    )
    (db-commit hdbc)
    (begin
        (display (db-get-error-str hdbc))
        (newline)
        (db-rollback hdbc)
    )
)
...
```

## Explanation

This example shows that if the application can successfully delete “John’s record” and update “Mary’s record,” it will commit the transaction specified by the connection. Otherwise, it prints out the error message and rolls back the transaction.

## make-connection-handle

### Syntax

```
(make-connection-handle)
```

### Description

**make-connection-handle** constructs the *connection handle*.

### Parameters

None.

### Return Values

**A handle**

Returns a connection-handle if successful, otherwise;

**Boolean**

Returns **#f** (false) if the function fails to create a connection-handle. Use **db-get-error-str** to retrieve the error message.

### Throws

None.

### Examples

```
(let ((hConnection (make-connection-handle)))
    (if (connection-handle? hConnection)
        (begin
            (display “Established a valid connection handle\n”)
        )
        (begin
            (display “Failed to get a connection handle: “)
            (display (db-get-error-str connection-handle))
            (newline)
        )
    )
)
```

### Explanation

The above example creates a connection handle variable called hConnection. The results are verified by using the **connection-handle?** function to check the type of the hConnection variable. If the results are a connection handle, then the message “Established a valid connection handle” is displayed. If the return value is not a connection handle, then the message “Failed to get a connection handle:” and the error string are displayed.

## statement-handle?

### Syntax

```
(statement-handle? any-variable)
```

### Description

**statement-handle?** determines whether or not the input argument is a statement handle datatype.

### Parameters

This function requires a single variable of any datatype.

### Return Values

**Boolean**

Returns **#t** (true) if the argument is a statement handle; otherwise, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

### Throws

None.

### Examples

```
(define hstmt (db-proc-bind hdbc “test”))
(if (not (statement-handle? hstmt))
    (display (db-get-error-str hdbc))
)
```

### Explanation

The above example creates a statement handle called hstmt, then it displays an error message if the newly defined hstmt is not a statement handle.

# 5.4 Static SQL Functions

The functions in this category control the e*Way's interaction with static SQL commands.

The static SQL functions are:

**db-sql-column-names** on page 126

**db-sql-column-types** on page 128

**db-sql-column-values** on page 129

**db-sql-execute** on page 131

**db-sql-fetch** on page 132

**db-sql-fetch-cancel** on page 133

**db-sql-format** on page 134

**db-sql-select** on page 136

## Static vs. Dynamic SQL Functions

Dynamic SQL statements are built and executed at run time versus Static SQL statements that are embedded within the program source code. Dynamic statements do not require knowledge of the complete structure on an SQL statement before building the application. This allows for run time input to provide information about the database objects to query.

The application can be written so that it prompts the user or scans a file for information that is not available at compilation time.

In Dynamic statements the four steps of processing an SQL statement take place at run time, but they are performed only once. Execution of the plan takes place only when EXECUTE is called. **Figure 34 on page 124** shows the difference between Dynamic SQL with immediate execution and Dynamic SQL with prepared execution.

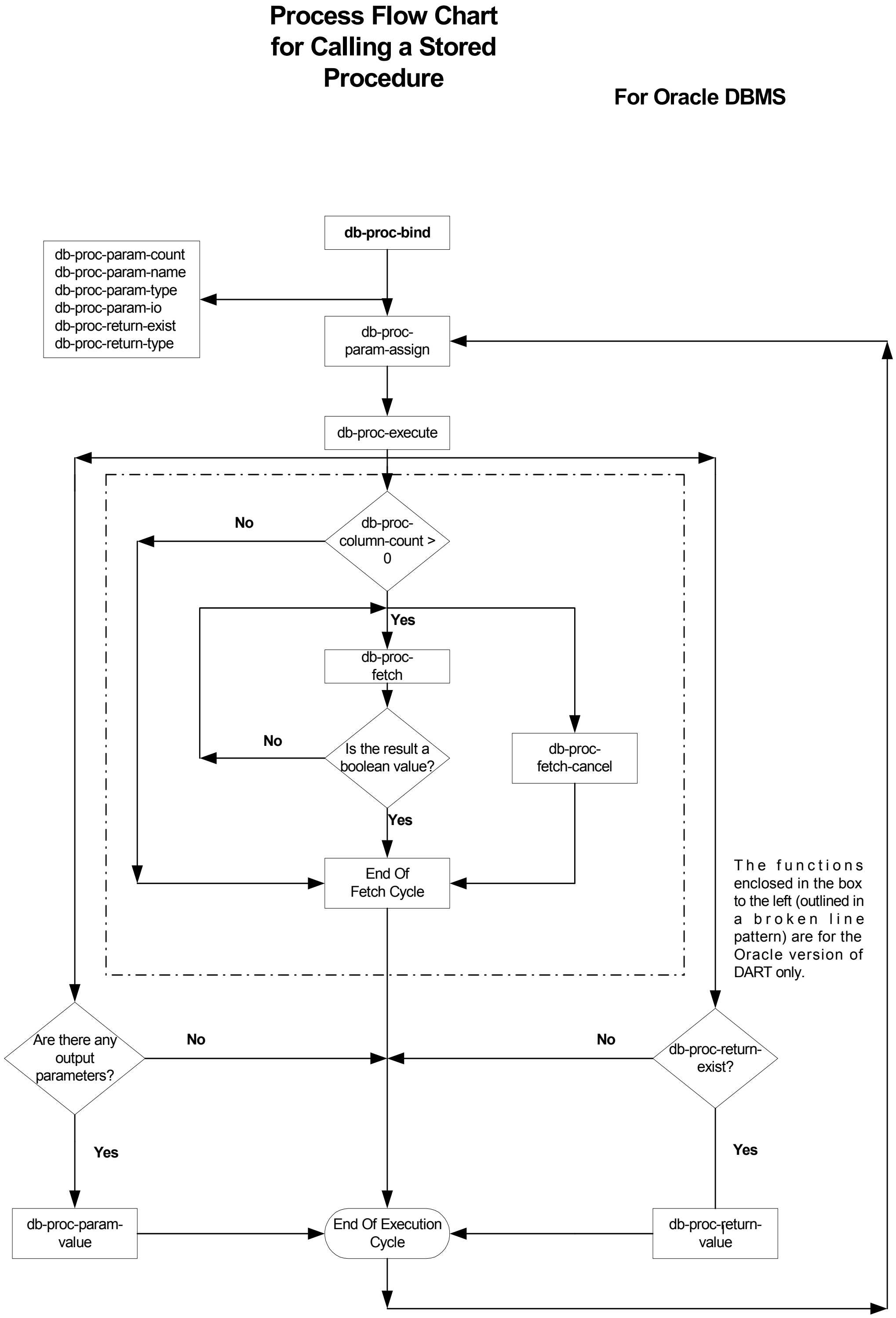
### Benefits of Dynamic SQL

Using dynamic SQL commands, an application can prepare a "generic" SQL statement once and execute it multiple times. Statements can also contain markers for parameter values to be supplied at execution time, so that the statement can be executed with varying inputs.

### Limitations of Dynamic SQL

The use of dynamic SQL commands has some significant limitations. A dynamic SQL implementation of an application generally performs worse than an implementation where permanent stored procedures are created and the client program invokes them with RPC (remote procedure call) commands.

**Figure 31** Calling a Stored Procedure (Oracle)



The functions enclosed in the box to the left (outlined in a broken line pattern) are for the Oracle version of DART only.

**Figure 32** Calling a Stored Procedure (Sybase)

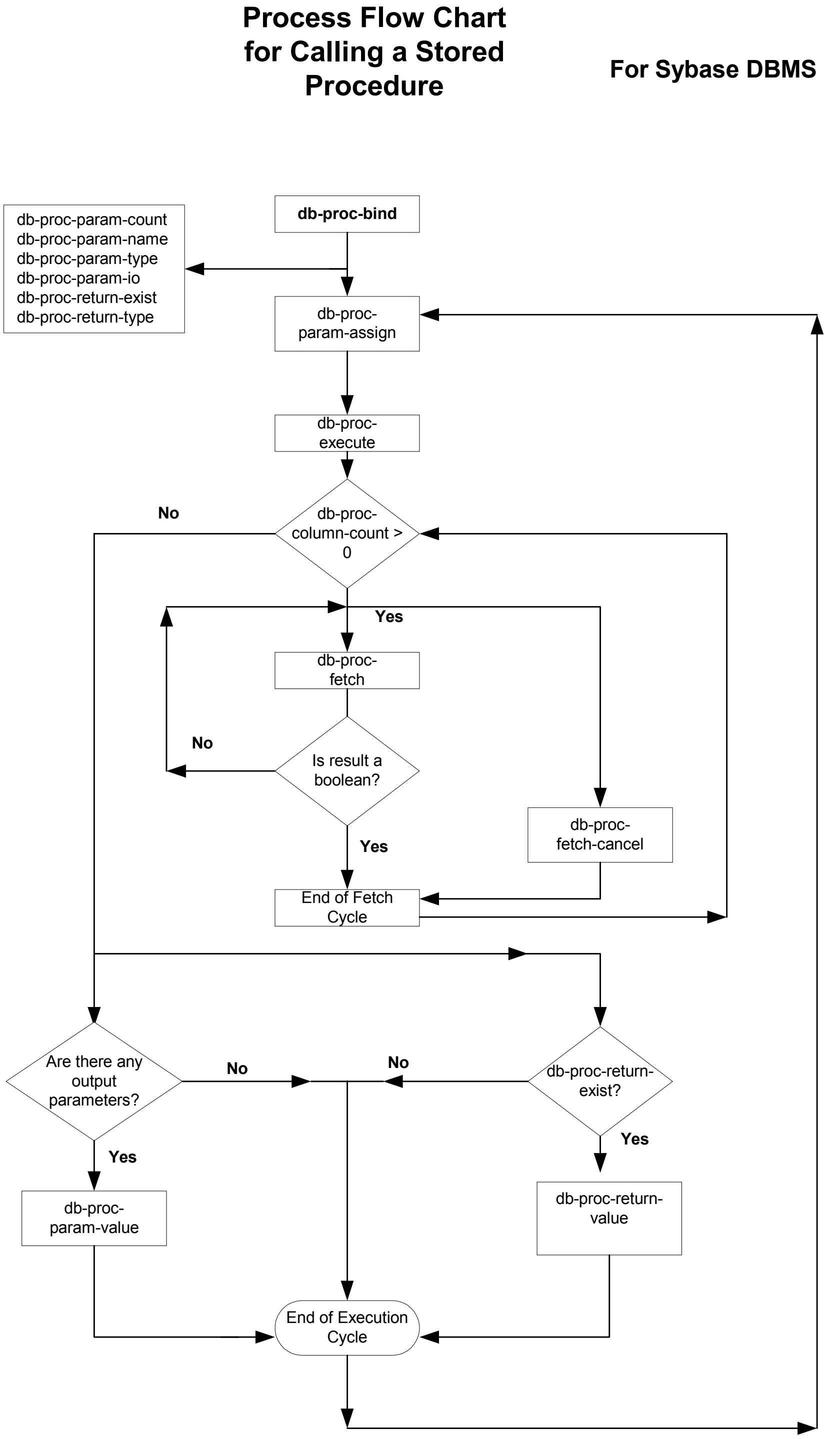


**Figure 33** Dynamic Statement Flow Chart

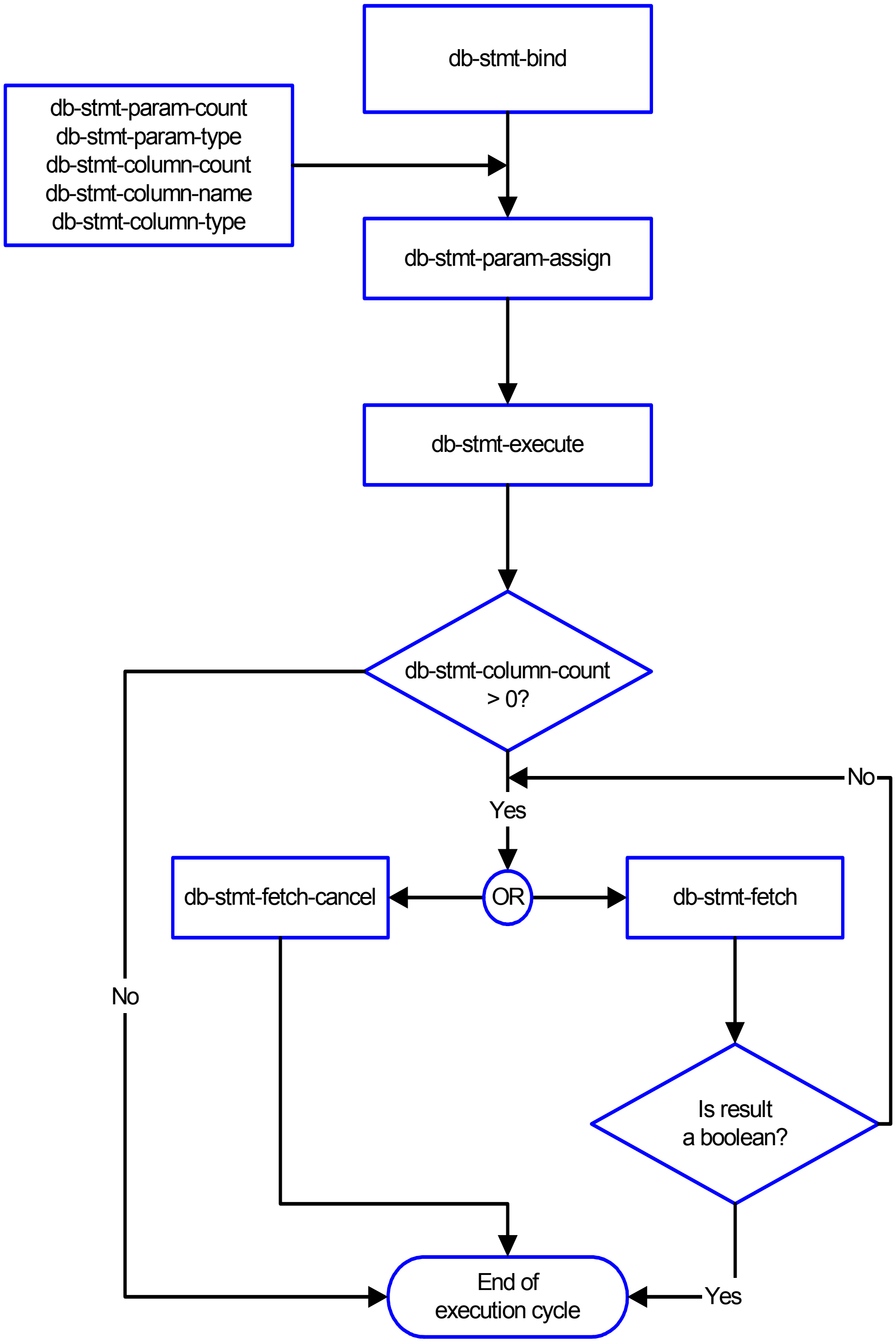
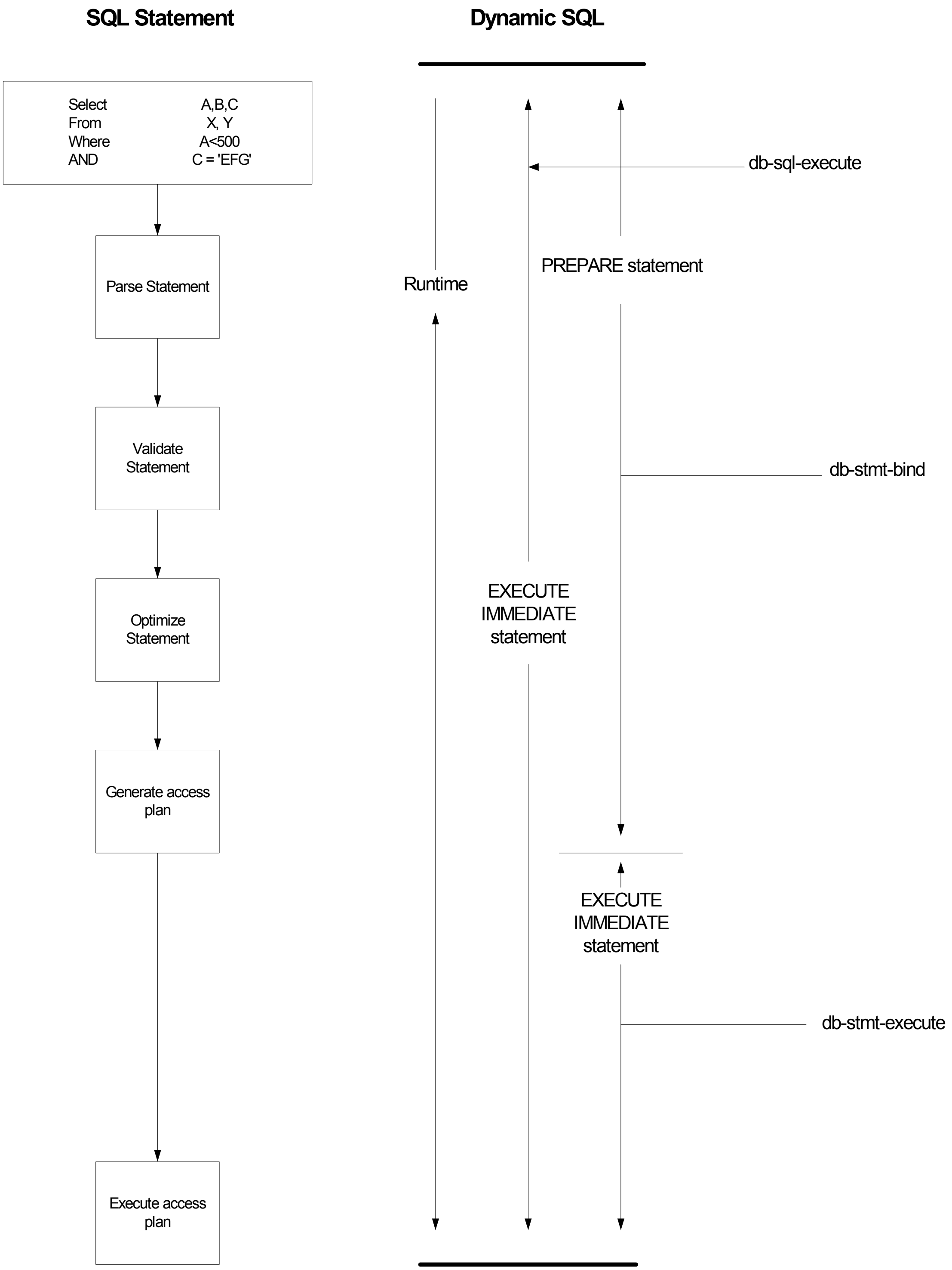


**Figure 34** Example of Dynamic SQL processing



## ODBC SQL Type Support

The following table shows the supported SQL datatypes and the corresponding native datatype for the database.

**Table 9** ODBC SQL Type Support

| SQL Type Name | SQL Datatype | Oracle Datatype |
|---|---|---|
| SQL_BIT | BIT | N/A |
| SQL_BINARY | BINARY (n) | N/A |
| SQL_VARBINARY | VARBINARY (n) | RAW (n) |
| SQL_CHAR | CHAR (n) | CHAR (n) |
| SQL_VARCHAR | VARCHAR (n) | VARCHAR2 (n) |
| SQL_DECIMAL | DECIMAL (p, s) | NUMBER (p, s) |
| SQL_NUMERIC | NUMERIC (p, s) | N/A |
| SQL_TINYINT | TINYINT | + |
| SQL_BIGINT | BIGINT | + |
| SQL_SMALLINT | SMALLINT | + |
| SQL_INTEGER | INTEGER | + |
| SQL_REAL | REAL | * |
| SQL_FLOAT | FLOAT(p) | FLOAT(b) |
| SQL_DOUBLE | DOUBLE PRECISION | FLOAT |
| SQL_DATE | DATE | N/A |
| SQL_TIME | TIME | N/A |
| SQL_TIMESTAMP | TIMESTAMP | DATE |
| SQL_LONGVARCHAR | LONG VARCHAR | LONG |
| SQL_LONGVARBINARY | LONG VARBINARY | LONG RAW |

*Oracle float (p) specifies a floating point number with precision range from 1 to 126.

+Oracle uses number (p) to define datatypes that span TININT, BIGINT, SMALLINT, and INTEGER. Oracle int type is internally mapped to NUMBER (38) which will be returned as SQL_DECIMAL.

**Note:** *All variable precision datatypes require precision values.*

SQL_DECIMAL and SQL_NUMERIC datatypes require specification of scale which indicates the number of digits to the right of the decimal point.

# db-sql-column-names

## Syntax

```
(db-sql-column-names connection-handle selection-name)
```

## Description

**db-sql-column-names** returns a vector of column names which are the result of an SQL SELECT statement identified by the parameter selection-name. This procedure can be called after a SQL SELECT statement has been issued successfully.

## Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| selection-name | string | The name that identifies the selection. |

## Return Values

### A string

This function returns a vector of column names in string format if successful.

### Boolean

If the selection-name string is unavailable for any reason, this function returns a **#f** (false). Use **db-get-error-str** to retrieve the error message.

## Throws

None.

## Examples

```
(define selection "select * from person where title='manager'")
(if (db-login hdbc "dsn" "uid" "pwd")
    (begin
        (if (db-sql-select hdbc "manager" selection)
            (begin
                (define name-array (db-sql-column-names hdbc 
"manager"))
                (if (vector? name-array)
                    (begin
                        (display "name of the first column: ")
                        (display (vector-ref name-array 0))
                        (newline)
                        ...
                    )
                    (begin
                        (display (db-get-error-str hdbc))
                        (newline)
                    )
                )
                (display (db-get-error-str hdbc))
                (newline)
            )
        )
        (if (db-alive hdbc)
            (begin
                    ...
                )
            )
            (db-logout hdbc)
        )
    )
```

### Explanation

This example shows that after issuing a successful SQL SELECT statement, the program will display the name of the first column.

## db-sql-column-types

### Syntax

```
(db-sql-column-types connection-handle selection-name)
```

### Description

**db-sql-column-types** returns a vector of column types which are the result of an SQL SELECT statement identified by the parameter selection-name. This procedure can be called after a SQL SELECT statement has been issued successfully. Refer to the description for *db-bind-proc* for a list of SQL-type names.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| selection-name | string | The name that identifies the selection. |

**A string**

This function returns a vector of column types in string format if successful.

**Boolean**

If the string type is unavailable for any reason, this function returns a **#f**. Use **db-get-error-str** to retrieve the error message.

### Throws

None.

### Examples

```
(define selection “select * from person where title= ‘manager’”)
        (define type-array (db-sql-column-types hdbc “manager”))
                (if (vector? type-array)
                    (begin
                        (display “type of the first column:”)
                        (display (vector-ref type-array 0))
                        (newline)
                        ...
                        ...
                    )
                    (begin
                        (display (db-get-error-str hdbc))
                        (newline)
                    )
                )
                (display (db-get-error-str hdbc))
            )
        )
        (if (db-alive hdbc)
            (begin
                ...
```

### Explanation

This example shows that after issuing a successful SQL SELECT statement, the program will display the first column type.

## db-sql-column-values

### Syntax

```
(db-sql-column-values  connection-handle  selection-name)
```

### Description

**db-sql-column-values** returns a vector of column values, which is the result of an SQL FETCH statement identified by the parameter selection-name. This procedure can be called after a SQL FETCH statement has been issued successfully.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| selection-name | string | The name that identifies the selection. |

### Return Values

**A string**

Returns a vector of SQL values in string format if successful.

**Boolean**

If the values string is unavailable for any reason, this function returns a **#f** (false).Use **db-get-error-str** to retrieve the error message.

### Throws

None.

### Examples

```
(define selection "select * from person where title= 'manager'")

(if (db-sql-select hdbc "manager" selection)
  (do ((result "") (value-array 0)) ((boolean? result))
    (set! result (db-sql-fetch hdbc "manager"))
    (if (not (boolean? reslt))
      (begin
        (set! value-array (db-sql-column-values hdbc "manager"))
        (do (
               (index 0 (+ index 1))
               (count (vector-length value-array))
            )
            ((= index count))
          (display (vector-ref value-array index))
          (display "\t")
        )
        (newline)
      )
      (if (not result) (display (db-get-error-str hdbc)))
    )
  )
  (begin
    (display (db-get-error-str hdbc))
    (newline)
  )
```

```
        )
```

### Explanation

This example shows that after issuing a successful SQL SELECT statement, the program will loop through a fetch cycle. Within each fetch loop, the program displays the value of each column in the same line, separated by a tab character.

### Notes

1. A successful **db-sql-fetch** call returns a string which contains the concatenation of all column values with the comma (,) character as the separator. Although this single string is suitable for display purposes, the user must parse the result string to retrieve the value of each column.
2. If the value of the column contains the comma (,) character, the user will be unable to differentiate the comma data from the comma separator. Therefore, **db-sql-column-values** returns the result as a vector of values in string type to allow the user to make use of the vector-ref function to retrieve the value of each column and avoid any parsing problem.

## db-sql-execute

### Syntax

```
(db-sql-execute connection-handle SQL-stmt)
```

### Description

**db-sql-execute** executes the specified SQL statement.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| SQL-stmt | string | The SQL statement being executed. |

### Return Values

**Boolean**

Returns **#t** (true) if successful; otherwise, returns **#f** (false).Use **db-get-error-str** to retrieve the error message.

### Throws

None.

### Examples

```
...
(if (db-login hdbc "Payroll" "James" "12345")
   (begin
        ...
        (if (db-sql-execute hdbc "insert into employee
values(‘John’...)")
            (db-commit hdbc)
         )
     )
    (display (db-get-error-str hdbc))
)
...
```

### Explanation

This example shows that if the application can successfully log into the data source “Payroll,” it will insert a record into the table “employee.”

### Notes

Use the **db-sql-select** function to execute a select statement.

The **db-sq.-execute** function can no longer be used to commit and roll back transactions. Instead, use **db-commit** or **db-rollback**.

***Note:*** *The Merant ODBC drivers limit the size of the SQL statement to 32 Kbyte.*

## db-sql-fetch

### Syntax

```
(db-sql-fetch  connection-handle  selection-name)
```

### Description

**db-sql-fetch** “fetches” the result of a SELECT statement. The statement handle is “free” after the function fetches the last record.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| selection-name | string | The name that identifies the selection. |

### Return Values

**A string**

Returns a comma, delimited string containing all the column values for the record.

**Boolean**

Returns **#t** (true) at the end of the “fetch cycle,” when no more records are available to “fetch"; otherwise, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

### Throws

None.

### Examples

```
...
(if (db-sql-select hdbc “GreaterThan25” “select * employee where age
> 25”)
    (begin
        (display (db-sql-fetch hdbc “GreaterThan25”))
        (newline)
        (db-sql-fetch-cancel hdbc “GreaterThan25”)
    )
    (begin
        (display (db-get-error-str hdbc))
        (newline)
    )
)
...
```

### Explanation

This example shows that the application selects the first record of employees who are older than age 25, by fetching the record once and cancelling the rest of the records.

### Notes

The return result is temporarily stored in RAM. The buffer is allocated when **db-sql-select** is called. The maximum size of the buffer is determined by the operating system.

# db-sql-fetch-cancel

## Syntax

```
(db-sql-fetch-cancel connection-handle selection-name)
```

## Description

**db-sql-fetch-cancel** closes the cursor associated with an SQL SELECT statement and cancels the fetch command. It also frees up the memory allocation for the selection.

## Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| selection-name | string | The name that identifies the selection. |

## Return Values

### Boolean

Returns **#t** (true) if successful; otherwise, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

## Throws

None.

## Examples

```
...
(if (db-sql-select hdbc "GreaterThan25" "select * employee where age
> 25")
    (begin
        (define result (db-sql-fetch hdbc "GreaterThan25"))
        (if (not (boolean? result))
            (db-sql-fetch-cancel hdbc "GreaterThan25")
            (if (not result)
                (begin
                    (display (db-get-error-str hdbc))
                    (newline)
                )
            )
        )
    )
    (begin
        (display (db-get-error-str hdbc))
        (newline)
    )
)
...
```

## Explanation

This example shows that the application selects the first record of employees who are older than age 25, by fetching the record once and cancelling the rest of the records.

## db-sql-format

### Syntax

```
(db-sql-format data-string  SQL-type)
```

### Description

**db-sql-format** returns a formatted string of the *data-string*, so it can be used in an SQL statement as a literal value of a corresponding *SQL-type*.

In the current implementation, only the SQL_CHAR, SQL_VARCHAR, SQL_DATE, SQL_TIME, and SQL_TIMESTAMP SQL-types will be formatted. If the *data-string* is an empty string, the procedure will return a NULL value for all SQL datatypes except SQL_CHAR and SQL_VARCHAR.

### Parameters

| Name | Type | Description |
|---|---|---|
| data-string | string | A data string to be used as a literal value in an SQL statement. |
| SQL-type | string | An SQL datatype string, i.e., SQL_VARCHAR. |

### Return Values

**A string**

Returns a formatted string used as a data value in an SQL statement.

### Throws

None.

### Examples

```
(define last-name (db-sql-format "O'Reilly" "SQL_VARCHAR"))
(define timestamp (db-sql-format "1998-02-19 12:34:56"
SQL_TIMESTAMP"))
(define sql-stmt (string-append "update employee set lastname =
"last-name ", MODIFYTIME = "timestamp "WHERE SSN = 123456789"))
(if (db-login hdbc "Payroll" "user" "password")
    (begin
        (if (db-sql-execute hdbc sql-stmt)
            (db-commit hdbc)
            (begin
                (display (db-get-error-str hdbc))
                (newline)
                (db-rollback hdbc)
            )
        )
        ...
        (db-logout hdbc)
    )
)
```

### Explanation

The example above illustrates how the program uses *db-sql-format* to format the last name and the timestamp and use the results as part of an SQL statement.

### Notes

1 For SQL_CHAR and SQL_VARCHAR (SQL datatypes) *db-sql-format* will place a single quotation mark (‘) before and after the <data-string>, and expand each single quotation mark in the <data-string> to two single quotation mark characters.

2 If you use the (timestamp) Monk built-in function to insert the timestamp to an Event Type Definition, you should specify the following format for it to be accepted by the **db-sql-format** function:

```
“%Y-%m-%d  %H:%M:%S”
```

For SQL_CHAR and SQL_VARCHAR (SQL datatypes) **db-sql-format** will place a single quotation mark (‘) before and after the *data-string*, and expand each single quotation mark in the *data-string* to two single quotation mark characters.

The following table shows the typical *data-string* and the corresponding results of the formatting for the OBDC e*Way.

**Table 10** SQL Statement Format

| SQL_type Value | Data_string Value | Formatted Result String |
|---|---|---|
| SQL_CHAR | This is a string | ‘This is a string.’ |
| SQL_VARCHAR | O’Reilly | ‘O’ ‘Reilly’ |
| SQL_DATE | 1998-02-19 | {d ‘1998-02019’} |
| SQL_DATE | 19980219 | {d ‘1998-02-19’} |
| SQL_TIME | 12 :34:56 | {t ‘12:34:56’} |
| SQL_TIME | 1234 | {t ‘12:34:00’} |
| SQL_TIMESTAMP | 1998-02-19 12:34:56.789 | {ts ‘1998-02-19 12:34:56.789’} |

## db-sql-select

### Syntax

```
(db-sql-select connection-handle selection-name SQL-statement)
```

### Description

**db-sql-select** executes an SQL SELECT statement.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| selection-name | string | The name that identifies the selection. |
| SQL-statement | string | The SELECT statement. |

### Return Values

**Boolean**

Returns **#t** (true) if successful; otherwise, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

### Throws

None.

### Examples

```
...
(if (db-sql-select hdbc “GreaterThan25” “select * employee where age
> 25”)
    (begin
        (display (db-sql-fetch hdbc “GreaterThan25”))
        (newline)
        (db-sql-fetch-cancel hdbc “GreaterThan25”)
    )
    (display (db-get-error-str hdbc))
)
...
```

### Explanation

This example shows that the application selects the first record of employees who are older than age 25, by fetching the records one at a time and cancelling the remainder of the return records.

***Note:*** *The Merant ODBC drivers limit the size of the SQL statement to 32 Kbyte.*

## 5.5 Dynamic SQL Functions

The function sin this category control the e*Way's interaction with dynamic SQL commands. For information about the differences between static and dynamic SQL functions, see **"Static vs. Dynamic SQL Functions" on page 120**.

The dynamic SQL functions are:

## db-stmt-bind

### Syntax

```
(db-stmt-bind connection-handle dynamic-SQL-statement)
```

### Description

**db-stmt-bind** binds the dynamic statement specified. The binary data type should be input or output parameters with hexadecimal format.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| dynamic-SQL-statement | string | The dynamic statement to be bound. |

### Return Values

**Statement handle**
The statement handle that identifies the dynamic statement specified.

**Boolean**
If unsuccessful, returns **#f** (false).Use **db-get-error-str** to retrieve the error message.

### Throws

None.

### Additional Information

If the user needs to input /output binary data in the raw (binary) format, they should use db-stmt-bind-binary.

### Notes

1 Oracle OCI API is unable to report the datatype for each bound parameter in a dynamic statement. All bound parameters will default to VARCHAR datatypes. This will allow Oracle to implicitly convert the data string of each parameter into the correct data value of the parameter at the execution of the dynamic statement.

2 If the user needs to select the long datatype column, the long column should appear at the end of the selection list.

## db-stmt-bind-binary

### Syntax

```
(db-stmt-bind-binary connection-handle dynamic-SQL-statement)
```

### Description

**db-stmt-bind-binary** binds the dynamic statement specified. The binary data type will be input and output with raw format.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| dynamic-SQL-statement | string | The dynamic statement to be bound. |

### Return Values

**Statement handle**
The statement handle that identifies the dynamic statement specified.

**Boolean**
If unsuccessful, returns **#f** (false).Use **db-get-error-str** to retrieve the error message.

### Throws

None.

# db-stmt-column-count

## Syntax

```
(db-stmt-column-count connection-handle statement-handle)
```

## Description

**db-stmt-column-count** returns the number of columns in the return result set.

## Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | string | The statement handle that identifies the dynamic statement specified. This is the handle produced by db-stmt-bind. |

## Return Values

**A number**

Returns a number greater than zero (0) when the record set is available.

**Boolean**

If no record set is available, the return value will be **#f** (false). Use **db-get-error-str** to retrieve the error message.

## Throws

None.

## db-stmt-column-name

### Syntax

```
(db-stmt-column-name connection-handle statement-handle index)
```

### Description

**db-stmt-column-name** returns the name string of the specified column in the result set.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the dynamic statement specified. |
| index | integer | An integer equal to -- 0 to db-stmt-column-count minus 1. |

### Return Values

**A string**

Returns the name string if successful.

**Boolean**

If unsuccessful, returns #f (false). Use db-get-error-str to retrieve the error message.

### Throws

None.

## db-stmt-column-type

### Syntax

```
(db-stmt-column-type connection-handle statement-handle index)
```

### Description

**db-stmt-column-type** returns the SQL datatype of the specified column in the record set.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the dynamic statement specified. |
| index | integer | An integer equal to -- 0 to db-stmt-column-count minus 1. |

### Return Values

**A string**

Returns a string of SQL datatype when successful.

**Boolean**

If unsuccessful, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

### Throws

None.

## db-stmt-execute

### Syntax

```
(db-stmt-execute connection-handle statement-handle)
```

### Description

**db-stmt-execute** executes the dynamic statement of a specified statement-handle.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | string | The statement handle that identifies the dynamic statement specified. This is the handle produced by db-stmt-bind. |

### Return Values

**Boolean**

If an error occurred, returns **#f** (false). Use **db-get-error-str** to obtain the error message.

### Throws

None.

## db-stmt-fetch

### Syntax

```
(db-stmt-fetch connection-handle statement-handle)
```

### Description

**db-stmt-fetch** retrieves the column values of the record set.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | string | The statement handle that identifies the dynamic statement specified. This is the handle produced by db-stmt-bind. |

### Return Values

**A Vector and a Boolean**

Returns a vector containing all the column values and at the end of the "fetch cycle" returns **#t** (true) when no more records are available to "fetch."

**Boolean**

If an error occurred, returns **#f** (false). Use **db-get-error-str** to obtain the error message.

### Throws

None.

## db-stmt-fetch-cancel

### Syntax

```
(db-stmt-fetch-cancel connection-handle statement-handle)
```

### Description

**db-stmt-fetch-cancel** terminates the current “fetch” cycle.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | string | The statement handle that identifies the dynamic statement specified. This is the handle produced by db-stmt-bind. |

### Return Values

**Boolean**

Returns **#t** (true) if successful; otherwise, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

### Throws

None.

# db-stmt-param-assign

## Syntax

```
(db-stmt-param-assign connection-handle statement-handle index value)
```

## Description

**db-stmt-param-assign** assigns the parameter and executes the dynamic statement of a specified parameter.

## Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | string | The statement handle that identifies the dynamic statement specified. This is the handle produced by db-stmt-bind. |
| index | integer | The number between 0 and db-stmt-param-count minus 1. |
| value | string | The value to be assigned to the parameter. |

## Return Values

### Boolean

If an error occurred, returns **#f** (false). Use **db-get-error-str** to obtain the error message.

## Throws

None.

## db-stmt-param-count

### Syntax

```
(db-stmt-param-count connection-handle  statement-handle)
```

### Description

**db-stmt-param-count** retrieves the number of parameters in the dynamic statement.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | string | The statement handle that identifies the dynamic statement specified. This is the handle produced by db-stmt-bind. |

### Return Values

**An Integer**
Returns a number, which represents the number of parameters for the dynamic statement specified, when successful.

**Boolean**
If unsuccessful, returns **#f** (false).Use **db-get-error-str** to retrieve the error message.

### Throws

None.

## db-stmt-param-type

### Syntax

```
(db-stmt-param-type connection-handle statement-handle index)
```

### Description

**db-stmt-param-type** retrieves the SQL datatype of the specified parameter.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | string | The statement handle that identifies the dynamic statement specified. This is the handle produced by db-stmt-bind. |
| index | integer | The number between 0 and db-stmt-param-count minus 1. |

### Return Values

**A string**

If successful, **db-stmt-param-type** returns a string which represents the SQL datatype.

**Boolean**

If an error occurred, returns **#f** (false). Use **db-get-error-str** to obtain the error message.

### Throws

None.

## db-stmt-row-count

### Syntax

```
(db-stmt-row-count connection-handle statement-handle index)
```

### Description

**db-stmt-row-count** returns the number of rows affected by the execution of the SQL statement.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the dynamic statement specified. |
| index | integer | An integer equal to -- 0 to db-stmt-column-count minus 1. |

### Return Values

**Boolean**

If unsuccessful, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

### Throws

None.

# 5.6 Stored Procedure Functions

The functions in this category control the e*Way's interaction with stored procedures.

The stored procedure functions are:

**db-proc-bind** on page 151

**db-proc-bind-binary** on page 152

**db-proc-column-count** on page 153

**db-proc-column-name** on page 155

**db-proc-column-type** on page 157

**db-proc-execute** on page 159

**db-proc-fetch** on page 161

**db-proc-fetch-cancel** on page 163

**db-proc-param-assign** on page 164

**db-proc-param-count** on page 166

**db-proc-param-io** on page 167

**db-proc-param-name** on page 168

**db-proc-param-type** on page 169

**db-proc-param-value** on page 170

**db-proc-return-exist** on page 172

**db-proc-return-type** on page 174

**db-proc-return-value** on page 176

### Benefits of Stored Procedures

When a stored procedure is created for an application, SQL statement compilation and optimization are performed once when the procedure is created. With a dynamic SQL application, compilation and optimization are performed every time the client program runs. A dynamic SQL implementation also incurs database space overhead because each instance of the client program must create separate compiled versions of the application's prepared statements. When you design an application to use stored procedures and RPC commands, all instances of the client program can share the same stored procedure.

# db-proc-bind

## Syntax

```
(db-proc-bind  connection-handle  procedure-name)
```

## Description

**db-proc-bind** binds the input/output parameters of the stored procedure specified.

## Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| procedure-name | string | The stored procedure to be bound. |

## Return Values

### Boolean

Returns a proc-handle if successful; otherwise, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

## Throws

None.

## Examples

```
(define hstmt (db-proc-bind hdbc "test")
(if (not (statement-handle? hstmt)
    (display "fail to bind stored procedure test\n")
)
```

## Notes

ODBC does not recognize any procedure inside the PACKAGE of the Oracle DBMS. Therefore, you cannot bind any procedure defined inside the PACKAGE of the Oracle DBMS.

The procedure name is limited to 30 characters.

## db-proc-bind-binary

### Syntax

```
(db-proc-bind-binary connection-handle  dynamic-SQL-statement)
```

### Description

**db-proc-bind-binary** binds the dynamic statement specified. The format of the input and output data is binary.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| dynamic-SQL-statement | string | The dynamic statement to be bound |

### Return Values

**A string**
Returns a statement-handle when successful; otherwise

**Boolean**
Returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

### Throws

None.

# db-proc-column-count

## Syntax

```
(db-proc-column-count  connection-handle  statement-handle)
```

## Description

**db-proc-column-count** retrieves the number of columns in the return result set.

## Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the stored procedure specified. This is the handle produced by db-proc-bind. |

## Return Values

**A number**
Returns a number greater than zero (0) when the record set is available.

**Boolean**
If no record set is available, the return value will be **#f** (false). Use **db-get-error-str** to retrieve the error message.

## Throws

None.

## Examples

```
(if (db-login hdbc dsn uid pwd)
  (begin
     (display “database login succeed !\n”)
     (define hstmt (db-proc-bind hdbc “TEST_PROC”))
     (if (statement-handle? hstmt)
       (if (db-proc-param-assign hdbc hstmt 0 “5”)
         (if (db-proc-execute hdbc hstmt)
           (begin
              (define col-count (db-proc-column-count))
              (if (and (number? col-count) (> col-count 0))
                (do ((i 0 (+ i 1))) ((= i col-count))
                   (display “column “)
                   (display (db-proc-column-name hdbc hstmt i))
                   (display “: type = ”)
                   (display (db-proc-column-type hdbc hstmt i))
                   (newline)
                )
                (display (db-get-error-str hdbc))
              )
              ...
              ...
           )
           (display (db-get-error-str hdbc))
       )
       (display (db-get-error-str hdbc))
```

```
            )
            (display (db-get-error-str hdbc))
         )
         ...
         ...
         )
          (display (db-get-error-str hdbc)
      )
```

### Notes

1. The Oracle procedure to return the result set is given here:

   `oracle_odbc.sql`

2. The ODBC configuration parameter to set up the return result set must show

   `ProcedureRetResult = 1`

   For more information see **"Sample .odbc.ini File" on page 16**.

# db-proc-column-name

## Syntax

```
(db-proc-column-name connection-handle statement-handle column-index)
```

## Description

**db-proc-column-name** retrieves the name string of the specified column in the result set.

## Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the stored procedure specified. This is the handle produced by db-proc-bind. |
| column-index | string | SQL datatype of the specified column in the results set --0 to db-proc-column-count minus 1. |

## Return Values

### A string

Returns the name string if successful.

### Boolean

If unsuccessful, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

## Throws

None.

## Examples

```
(if (db-login hdbc dsn uid pwd)
  (begin
     (display "database login succeed !\n")
     (define hstmt (db-proc-bind hdbc "TEST_PROC"))
     (if (statement-handle? hstmt)
       (if (db-proc-param-assign hdbc hstmt 0 "5")
         (if (db-proc-execute hdbc hstmt)
           (begin
              (define col-count (db-proc-column-count))
              (if (and (number? col-count) (> col-count 0))
                (do ((i 0 (+ i 1))) ((= i col-count))
                   (display "column ")
                   (display (db-proc-column-name hdbc hstmt i))
                   (display ": type = ")
                   (display (db-proc-column-type hdbc hstmt i))
                   (newline)
                )
                (display (db-get-error-str hdbc))
              )
              ...
              ...
           )
           (display (db-get-error-str hdbc))
         )
         (display (db-get-error-str hdbc))
     )
     (display (db-get-error-str hdbc))
  )
     ...
     ...
  )
 (display (db-get-error-str hdbc)
)
```

### Notes

Since the result set of a stored procedure is returned through the parameters of the PL/SQL table type, the name of the table type parameter will be returned.

# db-proc-column-type

## Syntax

```
(db-proc-column-type connection-handle statement-handle column-index)
```

## Description

**db-proc-column-type** retrieves the SQL datatype of the specified column in the record set.

## Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the stored procedure specified. This is the handle produced by db-proc-bind. |
| column-index | string | SQL datatype of the specified column in the record set --0 to db-proc-column-count minus 1. |

## Return Values

### A string

Returns a string of SQL datatype when successful.

### Boolean

If unsuccessful, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

## Throws

None.

## Examples

```
(if (db-login hdbc dsn uid pwd)
  (begin
     (display "database login succeed !\n")
     (define hstmt (db-proc-bind hdbc "TEST_PROC"))
     (if (statement-handle? hstmt)
       (if (db-proc-param-assign hdbc hstmt 0 "5")
         (if (db-proc-execute hdbc hstmt)
           (begin
              (define col-count (db-proc-column-count))
              (if (and (number? col-count) (> col-count 0))
                (do ((i 0 (+ i 1))) ((= i col-count))
                   (display "column ")
                   (display (db-proc-column-name hdbc hstmt i))
                   (display ": type = ")
                   (display (db-proc-column-type hdbc hstmt i))
                   (newline)
                )
                (display (db-get-error-str hdbc))
              )
              ...
              ...
           )
           (display (db-get-error-str hdbc))
         )
         (display (db-get-error-str hdbc))
       )
       (display (db-get-error-str hdbc))
     )
     ...
     ...
  )
   (display (db-get-error-str hdbc)
)
```

### Notes

Since the result set of the stored procedure is returned through the parameters of the PL/SQL table type, a PL/SQL table can only contain one standard Oracle datatype.

## db-proc-execute

### Syntax

```
(db-proc-execute  connection-handle  statement-handle)
```

### Description

**db-proc-execute** executes out a stored procedure.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the stored procedure specified. This is the handle produced by db-proc-bind. |

### Return Values

**Boolean**

Returns **#t** (true) if successful; otherwise, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

### Throws

None.

### Examples

```
(if (db-login hdbc dsn uid pwd)
  (begin
     (display “database login succeed !\n”)
     (define hstmt (db-proc-bind hdbc “TEST_PROC”))
     (if (statement-handle? hstmt)
       (if (db-proc-param-assign hdbc hstmt 0 “5”)
         (if (db-proc-execute hdbc hstmt)
             ...
             (display (db-get-error-str hdbc))
         )
         (display (db-get-error-str hdbc))
      )
       (display (db-get-error-str hdbc))
     )
     ...
     ...
     (db-logout hdbc)
  )
  (display (db-get-error-str hdbc))
)
```

### Notes

The default precision for number or real type is 38 for a column in the table. This is important when executing a stored procedure that retrieves values from that column in the table. The **db-proc-execute** function will fail if the exponential part of the value is larger than 38.

For example:

- 1.555E+38 is acceptable
- 1.55E+39 will prevent the successful retrieval of the column values

# db-proc-fetch

## Syntax

```
(db-proc-fetch connection-handle  statement-handle)
```

## Description

**db-proc-fetch** retrieves the column values of the record set.

## Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the stored procedure specified. This is the handle produced by db-proc-bind. |

## Return Values

### A vector and Boolean

Returns a vector containing all the column values and at the end of the "fetch cycle" returns **#t** (true) when no more records are available to "fetch."

### Boolean

If unsuccessful, this function returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

## Throws

None.

## Examples

```
(if (db-login hdbc dsn uid pwd)
   (begin
      (display "database login succeed !\n")
      (define hstmt (db-proc-bind hdbc "TEST_PROC"))
      (if (statement-handle? hstmt)
         (if (db-proc-param-assign hdbc hstmt 0 "5")
            (if (db-proc-execute hdbc hstmt)
               (begin
                  (define col-count (db-proc-column-count))
                  (if (and (number? col-count) (> col-count 0))
                    (do ((result (db-proc-fetch hdbc hstmt) (db-proc-
fetch hdbc hstmt)))
                       ((boolean? result) (begin (display result)
(newline)))
                       (display result
                       (newline)
                    )
                  )
                  ...
                  ...
               )
                             (display (db-get-error-str hdbc)
            )
                    (display (db-get-error-str hdbc)
         )
                 (display (db-get-error-str hdbc)
      )
      ...
      ...
   )
   (display (db-get-error-str hdbc)
)
```

### Notes

1 The Oracle procedure to return result set is given here:

```
oracle_odbc.sql
```

2 ODBC configuration parameter to set up the capability of return result set must show

```
ProcedureRetResult = 1
```

For more information see Sample **.odbc.ini** **“Sample .odbc.ini File” on page 16**.

# db-proc-fetch-cancel

## Syntax

```
(db-proc-fetch-cancel  connection-handle  statement-handle)
```

## Description

**db-proc-fetch-cancel** terminates the current “fetch” cycle.

## Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the stored procedure specified. This is the handle produced by db-proc-bind. |

## Return Values

### Boolean

Returns **#t** (true) if successful; otherwise, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

## Throws

None.

## Examples

```
(if (db-login hdbc dsn uid pwd)
   (begin
      (display “database login succeed !\n”)
      (define hstmt (db-proc-bind hdbc “TEST_PROC”))
      (if (statement-handle? hstmt)
          (if (db-proc-param-assign hdbc hstmt 0 “5”)
            (if (db-proc-execute hdbc hstmt)
               (begin
                  (define col-count (db-proc-column-count))
                  (if (and (number? col-count) (> col-count 0))
                    (db-proc-fetch-cancel hdbc hstmt)
               )
            )
            ...
            ...
          )
                    (display (db-get-error-str hdbc)
      )
            (display (db-get-error-str hdbc)
     )
          (display (db-get-error-str hdbc)
      )
   ...
   ...
   )
   (display (db-get-error-str hdbc)
)
```

## db-proc-param-assign

### Syntax

```
(db-proc-param-assign  connection-handle  statement-handle  param-
index  param-value)
```

### Description

**db-proc-param-assign** "assigns" the value of an IN or INOUT parameter and places that value into internal storage.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the dynamic statement specified. This is the handle produced by db-proc-bind. |
| param-index | integer | The number between 0 and db-proc-param-count minus 1. |
| param-value | string | The input value of the IN or INOUT parameter. |

### Return Values

**Boolean**

Returns **#t** (true) if successful; otherwise, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

### Throws

None.

### Examples

Scenario #1 — sample code for db-proc-param-assign

```
(if (db-login hdbc dsn uid pwd)
  (begin
     (display “database login succeed !\n”)
     (define hstmt (db-proc-bind hdbc “TEST_PROC”))
     (if (statement-handle? hstmt)
       (if (db-proc-param-assign hdbc hstmt 0 “5”)
         (if (db-proc-execute hdbc hstmt)
             ...
             (display (db-get-error-str hdbc))
         )
         (display (db-get-error-str hdbc))
      )
       (display (db-get-error-str hdbc))
     )
     ...
     ...
     (db-logout hdbc)
  )
  (display (db-get-error-str hdbc))
)
```

### Scenario #2 — sample code for db-proc-param-assign with multiple input arguments

```
(if (db-login hdbc dsn uid pwd)
  (begin
    (display “database login succeed !\n”)
    (define hstmt (db-proc-bind hdbc “TEST_PROC”))
    (if (statement-handle? hstmt)
        (if (and
              (db-proc-param-assign hdbc hstmt 0 “5”)
              (db-proc-param-assign hdbc hstmt 2 “O’REILLY”)
              (db-proc-param-assign hdbc hstmt 7 “1998-11-22
12:34:56”)
              (db-proc-param-assign hdbc hstmt 8 “1A2B78F0”)
            )
            (if (db-proc-execute hdbc hstmt)
             ...
             ...
            )
            (display (db-get-error-str hdbc))
        )
        (display (db-get-error-str hdbc))
    )
    ...
    ...
  )
  (display (db-get-error-str hdbc))
)
```

## Notes

The value for the param-value argument should be entered as a string, without enclosure in single quotation marks (‘) for SQL_CHAR and SQL_VARCHAR.

The literal value for SQL_BINARY and SQL_VARBINARY should be a hexadecimal string. Refer to Scenario #2 on above.

# db-proc-param-count

## Syntax

```
(db-proc-param-count  connection-handle  statement-handle)
```

## Description

**db-proc-param-count** retrieves the number of parameters in the stored procedure.

## Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the stored procedure specified. This is the handle produced by db-proc-bind. |

## Return Values

### A number

Returns a number, which represents the number of parameters for the stored procedure specified, when successful.

### Boolean

If the number is unavailable due to a problem within one of the arguments, the function returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

## Throws

None.

## Examples

```
(if (db-login hdbc dsn uid pwd)
    (begin
        (display “database login succeed !\n”)
        (define hstmt (db-proc-bind hdbc “TEST_PROC”))
        (if (statement-handle? hstmt)
          (do ((i 0 (+ i 1))) ((= i (db-proc-param-count hdbc hstmt)))
              (display “parameter ”)
               (display (db-proc-param-name hdbc hstmt i))
              (display “: type = ”)
              (display (db-proc-param-type hdbc hstmt i))
              (display “, io = ”)
              (display (db-proc-param-io hdbc hstmt i))
              (newline)
          )
          (display (db-get-error-str hdbc))
        )
        ...
        ...
    )
    (display (db-get-error-str hdbc))
)
```

# db-proc-param-io

## Syntax

```
(db-proc-param-io  connection-handle  statement-handle  param-index)
```

## Description

**db-proc-param-io** retrieves the IO type for the specified parameter.

## Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the dynamic statement specified. This is the handle produced by db-proc-bind. |
| param-index | integer | The number between 0 and db-proc-param-count minus 1. |

## Return Values

### A string

Returns an IO type string as **IN**, **OUT**, or **INOUT**

### Boolean

otherwise, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

## Throws

None.

## Examples

```
(if (db-login hdbc dsn uid pwd)
    (begin
        (display "database login succeed !\n")
        (define hstmt (db-proc-bind hdbc "TEST_PROC"))
        (if (statement-handle? hstmt)
          (do ((i 0 (+ i 1))) ((= i (db-proc-param-count hdbc hstmt)))
              (display "parameter ")
              (display (db-proc-param-name hdbc hstmt i))
              (display ": type = ")
              (display (db-proc-param-type hdbc hstmt i))
              (display ", io = ")
              (display (db-proc-param-io hdbc hstmt i))
              (newline)
           )
           (display (db-get-error-str hdbc))
         )
        ...
        ...
    )
    (display (db-get-error-str hdbc))
)
```

# db-proc-param-name

## Syntax

```
(db-proc-param-name  connection-handle  statement-handle  param-
index)
```

## Description

**db-proc-param-name** retrieves the name of the specified parameter.

## Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the dynamic statement specified. |
| param-index | integer | The number between 0 and db-proc-param-count minus 1. |

## Return Values

### A string

Returns the string containing the name of the parameter.

### Boolean

Returns **#f** (false) if unable to return the string containing the name of the parameter. Use **db-get-error-str** to retrieve the error message.

## Throws

None.

## Examples

```
(if (db-login hdbc dsn uid pwd)
    (begin
        (display "database login succeed !\n")
        (define hstmt (db-proc-bind hdbc "TEST_PROC"))
        (if (statement-handle? hstmt)
          (do ((i 0 (+ i 1))) ((= i (db-proc-param-count hdbc hstmt)))
              (display "parameter ")
              (display (db-proc-param-name hdbc hstmt i))
              (display ": type = ")
              (display (db-proc-param-type hdbc hstmt i))
              (display ", io = ")
              (display (db-proc-param-io hdbc hstmt i))
              (newline)
           )
           (display (db-get-error-str hdbc))
         )
        ...
        ...
    )
    (display (db-get-error-str hdbc))
)
```

# db-proc-param-type

## Syntax

```
(db-proc-param-type  connection-handle  statement-handle  param-
index)
```

## Description

**db-proc-param-type** retrieves the SQL datatype of the specified parameter.

## Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the dynamic statement specified. This is the handle produced by db-proc-bind. |
| param-index | integer | The number between 0 and db-proc-param-count minus 1. |

## Return Values

### A string

If successful, **db-proc-param-type** returns a string which represents the SQL datatype.

### Boolean

If an error occurred, returns **#f** (false). Use **db-get-error-str** to obtain the error message.

## Throws

None.

## Examples

```
(if (db-login hdbc dsn uid pwd)
    (begin
        (display "database login succeed !\n")
        (define hstmt (db-proc-bind hdbc "TEST_PROC"))
        (if (statement-handle? hstmt)
          (do ((i 0 (+ i 1))) ((= i (db-proc-param-count hdbc hstmt)))
              (display "parameter ")
              (display (db-proc-param-name hdbc hstmt i))
              (display ": type = ")
              (display (db-proc-param-type hdbc hstmt i))
              (display ", io = ")
              (display (db-proc-param-io hdbc hstmt i))
              (newline)
           )
           (display (db-get-error-str hdbc))
         )
        ...
        ...
    )
    (display (db-get-error-str hdbc))
)
```

## db-proc-param-value

### Syntax

```
(db-proc-param-value connection-handle statement-handle param-index)
```

### Description

**db-proc-param-value** is used to retrieve the value of the OUT or INOUT parameter.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the dynamic statement specified. This is the handle produced by db-proc-bind. |
| param-index | integer | The number between 0 and db-proc-param-count minus 1. |

### Return Values

**A string**

Returns a string which represents the value of the **OUT** or **INOUT** parameter.

**Boolean**

If unsuccessful, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

### Throws

None.

### Examples

```
(if (db-login hdbc dsn uid pwd)
   (begin
     (display “database login succeed !\n”)
     (define hstmt (db-proc-bind hdbc “TEST_PROC”))
     (if (statement-handle? hstmt)
        (if (db-proc-param-assign hdbc hstmt 0 “5”)
           (if (db-proc-execute hdbc hstmt)
              (begin
                 (define col-count (db-proc-column-count hdbc hstmt))
                 (if (and (number? col-count) (> col-count 0))
                    (do ((result (db-proc-fetch hdbc hstmt) (db-proc-
fetch hdbc hstmt)))
                       ((boolean? result))
                       (display result)
                       (newline)
                    )
                 )
                 (define prm-count (db-proc-param-count hdbc hstmt))
                 (do ((i 0 (+ i 1))) ((= i prm-count))
                    (if (not (equal? (db-proc-param-io hdbc hstmt i)
“IN”))
                       (begin
                          (display “output parameter ”)
                          (display (db-proc-param-name hdbc hstmt i))
                          (display “ = ”)
                         (display (db-proc-param-value hdbc hstmt i))
                          (newline)
                       )
                    )
                 )
                 ...
                 ...
              )
              (display (db-get-error-str hdbc))
           )
           (display (db-get-error-str hdbc))
        )
        (display (db-get-error-str hdbc))
     )
     ...
     ...
   )
     (display (db-get-error-str hdbc)
)
```

## db-proc-return-exist

### Syntax

```
(db-proc-return-exist connection-handle  statement-handle)
```

### Description

**db-proc-return-exist** determines whether or not the stored procedure has a return value.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the dynamic statement specified. This is the handle produced by db-proc-bind. |

### Return Values

**Boolean**

Returns **#t** (true) if a return value exists or **#f** (false) when no return value exists or an error occurs. Use **db-get-error-str** to retrieve the error message.

### Throws

None.

### Examples

```
(if (db-login hdbc dsn uid pwd)
  (begin
     (display "database login succeed !\n")
     (define hstmt (db-proc-bind hdbc "TEST_PROC"))
          (if (statement-handle? hstmt)
            (if (db-proc-param-assign hdbc hstmt 0 "5")
              (if (db-proc-execute hdbc hstmt)
                (begin
                   (define col-count (db-proc-column-count))
                   (if (and (number? col-count) (> col-count 0))
                    (do ((result (db-proc-fetch hdbc hstmt) (db-proc-
fetch hdbc hstmt)))
                       ((boolean? result))
                       (display result)
                       (newline)
                     )
                   )
                   (if (db-proc-return-exist hdbc hstmt)
                     (begin
                        (display "return type = ")
                        (display (db-proc-return-type hdbc hstmt))
                        (newline)
                        (display " return value = ")
                        (display (db-proc-return-value hdbc hstmt))
                        (newline)
                     )
                   )
                   . . .
                   ...
                )
                (display (db-get-error-str hdbc))
              )
              (display (db-get-error-str hdbc))
            )
            (display (db-get-error-str hdbc))
          )
     ...
     ...
  )
   (display (db-get-error-str hdbc)
)

)
                (display (db-get-error-str hdbc))
              )
              (display (db-get-error-str hdbc))
            )
            (display (db-get-error-str hdbc))
          )
     ...
     ...
  )
```

# db-proc-return-type

## Syntax

```
(db-proc-return-type  connection-handle  statement-handle)
```

## Description

**db-proc-return-type** determines the SQL datatype for the return value.

## Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the stored procedure specified. This is the handle produced by db-proc-bind. |

## Return Values

### A string

Returns a SQL datatype string, i.e., SQL_VARCHAR.

## Throws

None.

## Examples

```
(if (db-login hdbc dsn uid pwd)
  (begin
     (display "database login succeed !\n")
     (define hstmt (db-proc-bind hdbc "TEST_PROC"))
          (if (statement-handle? hstmt)
            (if (db-proc-param-assign hdbc hstmt 0 "5")
              (if (db-proc-execute hdbc hstmt)
                (begin
                   (define col-count (db-proc-column-count))
                   (if (and (number? col-count) (> col-count 0))
                    (do ((result (db-proc-fetch hdbc hstmt) (db-proc-
fetch hdbc hstmt)))
                       ((boolean? result))
                       (display result)
                       (newline)
                     )
                   )
                   (if (db-proc-return-exist hdbc hstmt)
                     (begin
                        (display "return type = ")
                        (display (db-proc-return-type hdbc hstmt))
                        (newline)
                        (display "return value = ")
                        (display (db-proc-return-value hdbc hstmt))
                        (newline)
                     )
                   )
                   ...
                   ...
                )
                (display (db-get-error-str hdbc))
              )
              (display (db-get-error-str hdbc))
            )
            (display (db-get-error-str hdbc))
          )
          ...
          ...
  )
   (display (db-get-error-str hdbc)
)
```

## db-proc-return-value

### Syntax

```
(db-proc-return-value  connection-handle  statement-handle)
```

### Description

**db-proc-return-value** retrieves the return value (return status) for the stored procedure.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the stored procedure specified. This is the handle produced by db-proc-bind. |

### Return Values

**A string**

Returns a string which represents the return value.

### Throws

None.

### Examples

```
(if (db-login hdbc dsn uid pwd)
  (begin
     (display "database login succeed !\n")
     (define hstmt (db-proc-bind hdbc "TEST_PROC"))
          (if (statement-handle? hstmt)
            (if (db-proc-param-assign hdbc hstmt 0 "5")
              (if (db-proc-execute hdbc hstmt)
                (begin
                   (define col-count (db-proc-column-count))
                   (if (and (number? col-count) (> col-count 0))
                    (do ((result (db-proc-fetch hdbc hstmt) (db-proc-
fetch hdbc hstmt)))
                       ((boolean? result))
                       (display result)
                       (newline)
                     )
                   )
                   (if (db-proc-return-exist hdbc hstmt)
                     (begin
                        (display "return type = ")
                        (display (db-proc-return-type hdbc hstmt))
                        (newline)
                        (display " return value = ")
                        (display (db-proc-return-value hdbc hstmt))
                        (newline)
                     )
                   )
                   ...
                   ...
```

```
                              )
                              (display (db-get-error-str hdbc))
                            )
                            (display (db-get-error-str hdbc))
                          )
                          (display (db-get-error-str hdbc))
                        )
                   ...
                   ...
                )
                 (display (db-get-error-str hdbc)
              )
```

### Notes

1 Stored procedures can return an integer value called a return status. This status indicates that the procedure completed successfully or shows the reason for failure. The SQL Server has a defined set of return values; or users can define their own return values.

2 The SQL Server reserves 0 to indicate a successful return, and negative values in the range of -1 to -99 are assigned to a listing of reasons for failure. Numbers 0 and -1 to -14 are in use currently (see below).

| Value | Meaning |
|---|---|
| 0 | procedure executed without error |
| -1 | missing object |
| -2 | datatype error |
| -3 | process was chosen as deadlock victim |
| -4 | permission error |
| -5 | syntax error |
| -6 | miscellaneous user error |
| -7 | resource error, such as out of space |
| -8 | non-fatal internal problem |
| -9 | system limit was reached |
| -10 | fatal internal inconsistency |
| -11 | fatal internal inconsistency |
| -12 | table or index is corrupt |
| -13 | database is corrupt |
| -14 | hardware error |

# 5.7 Message Event Functions

The functions in this category control the e*Way's message Event operations.

The message Event functions are:

## db-struct-call

### Syntax

```
(db-struct-call connection-handle statement-handle procedure-path)
```

### Description

**db-struct-call** calls the stored procedure using the value from the procedure-path node of the DART Event Type Definition, retrieves all procedure output and places this information into the DART Event Type Definition

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the stored procedure specified. This is the handle produced by db-proc-bind. |
| procedure-path | path | The absolute path to the procedure nodes in the Event Type Definition. |

### Return Values

**Boolean**

Returns **#t** (true) if successful; otherwise, returns **#f** (false).Use **db-get-error-str** to retrieve the error message.

## db-struct-execute

### Syntax

```
(db-struct-execute connection-handle statement-handle statement-path)
```

### Description

**db-struct-execute** calls the dynamic statement using the value from the *statement-path* node of the DART Event Type Definition, retrieves all dynamic statement output and places this information into the DART Event Type Definition.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| statement-handle | statement handle | The statement handle that identifies the stored procedure specified. This is the handle produced by db-stmt-bind. |
| statement-path | statement path | The absolute path to the statement nodes in the Event Type Definition. |

### Return Values

**Boolean**

Returns **#t** (true) when successful; otherwise **#f** (false).

### Throws

None.

## db-struct-fetch

### Syntax

```
(db-struct-fetch connection-handle table-path)
```

### Description

**db-struct-fetch** composes and executes an SQL FETCH statement according to the information and data carried under the table-path node of an Event Type Definition, and stores the return column values inside each of the column nodes.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| table-path | Event path | A table node in an Event Type Definition. |

### Return Values

**Path**

Returns the table path if the execution of the SQL FETCH statement is successful, or

**Boolean**

Returns **#t** (true) when the end of the fetch cycle is reached; otherwise, returns **#f** (false). Use **db-get-error-str** to retrieve error message.

### Throws

None.

### Examples

```
(define input-event-format-file-name “in.ssc”)
(define output-event-format-file-name “out.ssc”)
(load “in.ssc”)
(load “out.ssc”)
(define src-collapsed-nodes ‘())
(define dest-collapsed-nodes ‘())
(define collapsed-rules ‘())
(define xlate
    (let ((input ($make-event-map in-delm in-struct))
         (output ($make-event-map out-delm out-struct)))
         (lambda ($make-event-string)
             ($event-parse input $make-event-string)
             ($event-clear output)
             (begin
                 (if (db-struct-select hdbc ~output%out.dbo.table2)
                     (do ((result “”) ((boolean? result))
                         (set! result (db-struct-fetch hdbc
~output%out.dbo.table2))
                         (if (boolean? result))
                             (if (not result)
                                 (begin
                                     (display “db-struct-fetch
failed!\n”)
                                     (display (db-get-error-str hdbc))
                                     (newline)
                                 )
                                 (begin
                                     ...
                                 )
                             )
                             (begin
                                 (display result)
                                 (newline)
                             )
                         )
                     )
                 )
                 ...
                 (insert ”” ~output%out ””)
             )
         )
     )
)
```

### Explanation

The example above shows a typical code segment of a Collaboration Rule that uses the Event Type Definition. In this example, the output defined by **out.ssc** is an Event Type Definition. After clearing the output Event-string with the output Event Type Definition, the Collaboration procedure uses **db-struct-select** to issue an SQLSELECT statement based on the information carried under Event- path [~output%out.dbo.table2].

It repeatedly uses **db-struct-fetch** to issue the SQL FETCH statement and store the resulting column values inside each column node under the table path [~output%out.dbo.table2] until there are no more records to fetch.

## db-struct-insert

### Syntax

```
(db-struct-insert connection-handle table-path)
```

### Description

**db-struct-insert** composes and executes an SQL INSERT statement according to the information and data carried under the **table-path** node of an Event Type Definition.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| table-path | Event path | A table node in an Event Type Definition. |

### Return Values

**Boolean**

Returns **#t** (true) if the execution of the SQL INSERT statement is successful; otherwise, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

### Throws

**None.**

### Examples

```
(define input-event-format-file-name "in.ssc")
(define output-event-format-file-name "out.ssc")
(load "in.ssc")
(load "out.ssc")
(define src-collapsed-nodes '())
(define dest-collapsed-nodes '())
(define collapsed-rules '())
(define xlate
    (let ((input ($make-event-map in-delm in-struct))
          (output ($make-event-map out-delm out-struct)))
         (lambda ($make-event-string)
             ($event-parse input $make-event-string)
             ($event-clear output)
             (begin
                 (if (db-struct-insert hdbc ~input%in.dbo.table2)
                     (begin
                         ...
                     )
                     (begin
                         (display (db-get-error-str hdbc))
                         (newline)
                     )
                 )
                 ...
                 (insert "" ~output%out "")
             )
         )
    )
```

```
        )
```

### Explanation

The example above shows a typical code segment of a Collaboration Rule that uses the Event Type Definition. In this example, the input Event Definition defined by “in.ssc” is an Event Type Definition. After parsing the input Event-string with the input Event Definition, the Collaboration procedure uses **db-struct-insert** to issue an SQL INSERT statement based on the information carried under Event path [~input%in.dbo.table2].

## db-struct-select

### Syntax

```
(db-struct-select  connection-handle  table-path  where-clause)
```

### Description

**db-struct-select** composes and executes an SQL SELECT statement according to the information and data carried under the table-path node of an Event Type Definition.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| table-path | Event path | A table node in an Event Type Definition. |
| where-clause | string | The where clause of the SQL SELECT statement. |

### Return Values

**Boolean**

Returns **#t** (true) if the execution of the SQL SELECT statement is successful; otherwise, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

### Throws

None.

### Examples

```
(define input-event-format-file-name “in.ssc”)
(define output-event-format-file-name “out.ssc”)
(load “in.ssc”)
(load “out.ssc”)
(define src-collapsed-nodes ‘())
(define dest-collapsed-nodes ‘())
(define collapsed-rules ‘())
(define xlate
    (let ((input ($make-event-map in-delm in-struct))
          (output ($make-event-map out-delm out-struct)))
         (lambda ($make-event-string)
             ($event-parse input $make-event-string)
             ($event-clear output)
             ($event-parse output (event->string output))
             (begin
                (if (db-struct-select hdbc ~output%out.dbo.table2 “ID
= 5”)
                      (begin
                         (db-struct-fetch hdbc ~output%out.dbo.table2)
                          ...
                             (db-sql-fetch-cancel hdbc “dbo.table2”)
                      )
                      (begin
                          (display (db-get-error-str hdbc))
                        (newline)
                      )
```

```
                            )
                            ...
                            (insert "" ~output%out "")
                        )
                    )
               )
          )
```

## Explanation

The example above shows a typical code segment of a Collaboration Rules file that uses the Event Type Definition. In this example, the output defined by **out.ssc** is an Event Type Definition. After clearing the output Event-string, the Collaboration procedure uses **db-struct-select** to issue an SQL SELECT statement based on the information carried under the Event-path [~output%out.dbo.table2]. The selection was cancelled by **db-sql-fetch-cancel** with "dbo.table2" as the selection name.

## Notes

1. Both **db-struct-select**, and **db-struct-fetch** use the same algorithm to generate the selection name for the **db-sql-select** and **db-sql-fetch** procedure call. If the table path is a table node under an owner (schema) node the selection name will be **owner.table**.
2. If the table path does not have an owner node above it, the selection name will be **table**. You must issue a **db-sql-fetch-cancel** call with either **owner.table** or **table** as the selection name, if you want to cancel the selection.

## db-struct-update

### Syntax

```
(db-struct-update connection-handle table-path where-clause)
```

### Description

**db-struct-update** composes and executes an SQL UPDATE statement according to the information and data carried under the table-path node of an Event Type Definition.

### Parameters

| Name | Type | Description |
|---|---|---|
| connection-handle | connection handle | A connection handle to the database. |
| table-path | Event path | A table node in an Event Type Definition. |
| where-clause | string | The where clause of the SQL SELECT statement. |

### Return Values

**Boolean**

Returns **#t** (true) if the execution of the SQL UPDATE statement is successful; otherwise, returns **#f** (false). Use **db-get-error-str** to retrieve the error message.

### Throws

None.

### Examples

```
(define input-event-format-file-name "in.ssc")
(define output-event-format-file-name "out.ssc")
(load "in.ssc")
(load "out.ssc")
(define src-collapsed-nodes '())
(define dest-collapsed-nodes '())
(define collapsed-rules '())
(define xlate
    (let ((input ($make-event-map in-delm in-struct))
          (output ($make-event-map out-delm out-struct)))
         (lambda ($make-event-string)
             ($event-parse input $make-event-string)
             ($event-clear output)
             (begin
                (if (db-struct-update hdbc ~input%in.dbo.table2 "ID =
5")
                      (begin
                          ...
                      )
                      (begin
                          (display (db-get-error-str hdbc))
                          (newline)
                      )
                 )
                 ...
                 (insert "" ~output%out "")
```

```
                       )
                   )
              )
          )
```

### Explanation

The example above shows a typical code segment of a Collaboration Rule that uses the Event Type Definition. In this example, the input defined by **in.ssc** is an Event Type Definition. After parsing the input Event-string with the input Event Type Definition, the Collaboration procedure uses **db-struct-update** to issue an SQL UPDATE statement based on the information carried under the Event-path [~input%in.dbo.table2].

# 5.8 Sample Monk Scripts

This section includes sample Monk scripts which demonstrate how to use the ODBC e*Way's Monk functions. These Monk scripts demonstrate the following activities:

- **"Initializing Monk Extensions" on page 190**
- **"Calling Stored Procedures" on page 191**
- **"Inserting Records with Dynamic SQL Statements" on page 193**
- **"Updating Records with Dynamic SQL Statements" on page 195**
- **"Selecting Records with Dynamic SQL Statements" on page 197**
- **"Deleting Records with Dynamic SQL Statements" on page 199**
- **"Inserting a Binary Image to a Database" on page 200**
- **"Retrieving an Image from a Database" on page 203**
- **"Common Supporting Routines" on page 205**

### 5.8.1 Initializing Monk Extensions

The sample script shows how to initialize the Monk extensions. This function is used by many of the other sample Monk scripts shown in this chapter.

To use this sample script in an actual implementation, modify the following values:

- **EGATE** – This designates the location of the e*Gate client.
- **dsn** – This is he name of the data source.
- **uid** – This is the user name.
- **pwd** – This is the login password.

```
;demo-init.monk

(define EGATE "/eGate/client")

; routine to load DART Monk extension
(define (load-library extension)
   (define filename (string-append EGATE "/bin/" extension))
   (if (file-exists? filename)
      (load-extension filename)
      (begin
         (display (string-append "File " filename " does not
exist.\n"))
         (abort filename)
      )
   )
)

(load-library "stc_monkext.dll")

;;
;; define STCDB variables, data source, user ID, and password
;;

(define STCDB "ORACLE8")

(load-library "stc_dbmonkext.dll")

(define dsn "database")
(define uid "Administrator")
(define pwd (encrypt-password uid "password"))
```

### 5.8.2 Calling Stored Procedures

This script gives an example of calling Stored Procedures. See **"Stored Procedure Functions" on page 150** for more details.

```
;demo-proc-execute.monk

; load Monk database extension
(load "demo-init.monk")
(load "demo-common.monk")

; call stored procedure and display results
(define (execute-procedure hdbc hstmt)
   (let ((prm-count (db-proc-param-count hdbc hstmt)))
      (if (db-proc-execute hdbc hstmt)
         (begin
            (do ((col-count (db-proc-column-count hdbc hstmt) (db-
proc-column-count hdbc hstmt)))
                ((or (not (number? col-count)) (= col-count 0)))
               (display-proc-column-property hdbc hstmt col-count)
               (display-proc-column-value hdbc hstmt col-count)
            )
           (display-proc-parameter-output-value hdbc hstmt prm-count)
            (if (db-proc-return-exist hdbc hstmt)
               (begin
                  (display "return: value = ")
                  (display (db-proc-return-value hdbc hstmt))
                  (newline)
               )
            )
         )
         (display (db-get-error-str hdbc))
      )
   )
)

; make new connection handle
(define hdbc (make-connection-handle))
(if (not (connection-handle? hdbc))
   (display (db-get-error-str hdbc))
)

(if (db-login hdbc dsn uid pwd)
   (begin
      (display "\ndatabase login succeed !\n")

      ; bind the stored procedure
      (define hstmt1 (bind-procedure hdbc "PERSONNEL.GET_EMPLOYEES"))

      ; call the stored procedure if the binding is successful
      (if (statement-handle? hstmt1)
         (begin
            (display "call PERSONNEL.GET_EMPLOYEES to get all sales
...\n\n")
            (if (and
                  (db-proc-param-assign hdbc hstmt1 0 "30")
                  (db-proc-param-assign hdbc hstmt1 1 "10")
                )
               (execute-procedure hdbc hstmt1)
               (display (db-get-error-str hdbc))
            )
         )
      )
```

```
               (if (not (db-logout hdbc))
                  (display (db-get-error-str hdbc))
               )
            )
            (display (db-get-error-str hdbc))
         )
```

### 5.8.3 Inserting Records with Dynamic SQL Statements

```
;demo-stmt-insert.monk

; load Monk database extension
(load "demo-init.monk")
(load "demo-common.monk")

; execute dynamic statement and display results
(define (execute-statement hdbc hstmt)
   (if (db-stmt-execute hdbc hstmt)
      (begin
         (display-stmt-row-count hdbc hstmt)
         #t
      )
      #f
   )
)

; make new connection handle
(define hdbc (make-connection-handle))
(if (not (connection-handle? hdbc))
   (display (db-get-error-str hdbc))
)

(define stmt1 "INSERT INTO SCOTT.BONUS SELECT ENAME, JOB, SAL, COMM
FROM SCOTT.EMP WHERE DEPTNO = ?")

(if (db-login hdbc dsn uid pwd)
   (begin
      (display "\ndatabase login succeed !\n")

      ; bind the dynamic statement
      (define hstmt1 (bind-statement hdbc stmt1))

      ; assign parameter and execute the dynamic statement
      (if (statement-handle? hstmt1)
         (begin
            (display "\nInsert accounting department into bonus table
...\n")
            (if (db-stmt-param-assign hdbc hstmt1 0 "10")
               (if (execute-statement hdbc hstmt1)
                  (begin
                     (display "\nCommit the insertions ...\n")
                     (if (not (db-commit hdbc))
                        (display (db-get-error-str hdbc))
                     )
                  )
                  (display (db-get-error-str hdbc))
               )
               (display (db-get-error-str hdbc))
            )

            (display "\nInsert sales department into bonus table
...\n")
            (if (db-stmt-param-assign hdbc hstmt1 0 "20")
               (if (execute-statement hdbc hstmt1)
                  (begin
                     (display "\nCommit the insertions ...\n")
                     (if (not (db-commit hdbc))
                        (display (db-get-error-str hdbc))
                     )
                  )
                  (display (db-get-error-str hdbc))
               )
               (display (db-get-error-str hdbc))
            )
         )
      )

      (if (not (db-logout hdbc))
         (display (db-get-error-str hdbc))
      )
   )
   (display (db-get-error-str hdbc))
)
```

### 5.8.4 Updating Records with Dynamic SQL Statements

```
;demo-stmt-update.monk

; load Monk database extension
(load "demo-init.monk")
(load "demo-common.monk")

; execute dynamic statement and display results
(define (execute-statement hdbc hstmt)
   (if (db-stmt-execute hdbc hstmt)
      (begin
         (display-stmt-row-count hdbc hstmt)
         #t
      )
      #f
   )
)

; make new connection handle
(define hdbc (make-connection-handle))
(if (not (connection-handle? hdbc))
   (display (db-get-error-str hdbc))
)

(define stmt1 "UPDATE SCOTT.BONUS SET COMM = ? WHERE JOB = ?")

(if (db-login hdbc dsn uid pwd)
   (begin
      (display "\ndatabase login succeed !\n")

      ; bind the dynamic statement
      (define hstmt1 (bind-statement hdbc stmt1))

      ; assign parameter and execute the dynamic statement
      (if (statement-handle? hstmt1)
         (begin
            (display "\nUpdate commission of manager ...\n")
            (if
               (and
                  (db-stmt-param-assign hdbc hstmt1 0 "10")
                  (db-stmt-param-assign hdbc hstmt1 1 "MANAGER")
               )
               (if (execute-statement hdbc hstmt1)
                  (begin
                     (display "\nCommit the updates ...\n")
                     (if (not (db-commit hdbc))
                        (display (db-get-error-str hdbc))
                     )
                  )
                  (display (db-get-error-str hdbc))
               )
               (display (db-get-error-str hdbc))
            )

            (display "\nUpdate commission of clerk ...\n")
            (if
               (and
                  (db-stmt-param-assign hdbc hstmt1 0 "20")
                  (db-stmt-param-assign hdbc hstmt1 1 "CLERK")
               )
               (if (execute-statement hdbc hstmt1)
                  (begin
                     (display "\nCommit the updates ...\n")
                     (if (not (db-commit hdbc))
                        (display (db-get-error-str hdbc))
                     )
                  )
                  (display (db-get-error-str hdbc))
               )
               (display (db-get-error-str hdbc))
            )
         )
      )

      (if (not (db-logout hdbc))
         (display (db-get-error-str hdbc))
      )
   )
   (display (db-get-error-str hdbc))
)
```

### 5.8.5 Selecting Records with Dynamic SQL Statements

```
;demo-stmt-select.monk

; load Monk database extension
(load "demo-init.monk")
(load "demo-common.monk")

; execute dynamic statement and display results
(define (execute-statement hdbc hstmt)
   (if (db-stmt-execute hdbc hstmt)
      (begin
         (display-stmt-column-value hdbc hstmt)
         #t
      )
      #f
   )
)

; make new connection handle
(define hdbc (make-connection-handle))
(if (not (connection-handle? hdbc))
   (display (db-get-error-str hdbc))
)

(define stmt1 "SELECT EMPNO, ENAME, JOB FROM SCOTT.EMP WHERE JOB = ?")
(define stmt2 "SELECT ENAME, DNAME, JOB, HIREDATE FROM SCOTT.EMP,
SCOTT.DEPT WHERE EMP.DEPTNO = DEPT.DEPTNO AND DEPT.DNAME = ?")

(if (db-login hdbc dsn uid pwd)
   (begin
      (display "\ndatabase login succeed !\n")

      ; bind the dynamic statements
      (define hstmt1 (bind-statement hdbc stmt1))
      (define hstmt2 (bind-statement hdbc stmt2))

      ; assign parameter and execute the dynamic statement
      (if (statement-handle? hstmt1)
         (begin
            (display "\nList all salesman ...\n\n")
            (if (db-stmt-param-assign hdbc hstmt1 0 "SALESMAN")
               (if (not (execute-statement hdbc hstmt1))
                  (display (db-get-error-str hdbc))
               )
               (display (db-get-error-str hdbc))
            )
            (display "\nList all manager ...\n\n")
            (if (db-stmt-param-assign hdbc hstmt1 0 "MANAGER")
               (if (not (execute-statement hdbc hstmt1))
                  (display (db-get-error-str hdbc))
               )
               (display (db-get-error-str hdbc))
            )
         )
      )

      (if (statement-handle? hstmt2)
         (begin
            (display "\nList employee of accounting department
...\n\n")
            (if (db-stmt-param-assign hdbc hstmt2 0 "ACCOUNTING")
               (if (not (execute-statement hdbc hstmt2))
                  (display (db-get-error-str hdbc))
               )
               (display (db-get-error-str hdbc))
            )
         )
         (display (db-get-error-str hdbc))
      )

      (if (not (db-logout hdbc))
         (display (db-get-error-str hdbc))
      )
   )
   (display (db-get-error-str hdbc))
)
```

## 5.8.6 Deleting Records with Dynamic SQL Statements

```
;demo-stmt-delete.monk

; load Monk database extension
(load "demo-init.monk")
(load "demo-common.monk")

; execute dynamic statement and display results
(define (execute-statement hdbc hstmt)
   (if (db-stmt-execute hdbc hstmt)
      (begin
         (display-stmt-row-count hdbc hstmt)
         #t
      )
      #f
   )
)

; make new connection handle
(define hdbc (make-connection-handle))
(if (not (connection-handle? hdbc))
   (display (db-get-error-str hdbc))
)

(define stmt1 "DELETE FROM SCOTT.BONUS WHERE ENAME IS NOT NULL")

(if (db-login hdbc dsn uid pwd)
   (begin
      (display "\ndatabase login succeed !\n")

      ; bind the dynamic statement
      (define hstmt1 (bind-statement hdbc stmt1))

      ; assign parameter and execute the dynamic statement
      (if (statement-handle? hstmt1)
         (begin
            (display "\nDelete records from scott.bonus table ...\n")
            (if (execute-statement hdbc hstmt1)
               (begin
                  (display "\nCommit the deletions ...\n")
                  (if (not (db-commit hdbc))
                     (display (db-get-error-str hdbc))
                  )
               )
               (display (db-get-error-str hdbc))
            )
         )
      )

      (if (not (db-logout hdbc))
         (display (db-get-error-str hdbc))
      )
   )
   (display (db-get-error-str hdbc))
)
```

### 5.8.7 Inserting a Binary Image to a Database

This sample shows how to insert a Binary Image into a Database. It uses both Static and Dynamic SQL functions. See **“Static SQL Functions” on page 120** and **“Dynamic SQL Functions” on page 137** for more details.

```
;demo-image-insert.monk

; load Monk database extension
(load "demo-init.monk")
(load "demo-common.monk")

(define (query-exist hdbc hstmt id)
   (let ((rec-count 0) (result '#()))
      (if (db-stmt-param-assign hdbc hstmt 0 id)
         (if (db-stmt-execute hdbc hstmt)
            (begin
              (set! result (vector-ref (db-stmt-fetch hdbc hstmt) 0))
               (set! rec-count (string->number result))
               (set! result (db-stmt-fetch-cancel hdbc hstmt))
               (if (> rec-count 0)
                  (begin
                     (display "image already exist\n")
                     #t
                  )
                  #f
               )
            )
            (begin
               (display (db-get-error-str hdbc))
               #f
            )
         )
         (begin
            (display (db-get-error-str hdbc))
            #f
         )
      )
   )
)

(define (execute-statement hdbc hstmt)
   (let ((col-count (db-stmt-column-count hdbc hstmt)) (row-count 0))
      (if (db-stmt-execute hdbc hstmt)
         (begin
            (if (> col-count 0)
               (if (not (display-stmt-column-value hdbc hstmt col-
count))
                  (display (db-get-error-str hdbc))
               )
            )
            (set! row-count (db-stmt-row-count hdbc hstmt))
            (if (boolean? row-count)
               (display (db-get-error-str hdbc))
               (display (string-append "number of image insert = "
(number->string row-count) "\n"))
            )
            (newline)
            #t
         )
         #f
      )
   )
```

```
)

(define (bind-image-statement hdbc stmt)
   (let ((hstmt (db-stmt-bind-binary hdbc stmt)))
      (display (string-append "\nDynamic statement : " stmt "\n"))
      (if (statement-handle? hstmt)
         (begin
;            (db-stmt-param-bind hdbc hstmt 0 "SQL_INTEGER" 4 0)
;            (db-stmt-param-bind hdbc hstmt 1 "SQL_VARCHAR" 20 0)
;            (db-stmt-param-bind hdbc hstmt 2 "SQL_VARCHAR" 10 0)
;            (db-stmt-param-bind hdbc hstmt 3 "SQL_INTEGER" 38 0)
;            (db-stmt-param-bind hdbc hstmt 4 "SQL_INTEGER" 38 0)
;            (db-stmt-param-bind hdbc hstmt 5 "SQL_INTEGER" 10 0)
            (db-stmt-param-bind hdbc hstmt 6 "SQL_LONGVARBINARY"
2000000 0)
            (define prm-count (db-stmt-param-count hdbc hstmt))
            (display-stmt-parameter-property hdbc hstmt prm-count)

            (define col-count (db-stmt-column-count hdbc hstmt))
            (display-stmt-column-property hdbc hstmt col-count)
          )
          (display (db-get-error-str hdbc))
      )
      hstmt
   )
)


(define image1-id "7100")
(define image1-name "Coast")
(define image1-type "JPEG")
(define image1-width "1280")
(define image1-height "1024")
(define image1-file (string-append image1-name ".jpg"))

(define image-port (open-input-file image1-file))
(define image1-data (read image-port 1000000))
(close-port image-port)
(define image1-size (number->string (string-length image1-data)))

(define image2-id "7200")
(define image2-name "Launch")
(define image2-type "JPEG")
(define image2-width "2000")
(define image2-height "1600")
(define image2-file (string-append image2-name ".jpg"))

(define image-port (open-input-file image2-file))
(define image2-data (read image-port 2000000))
(close-port image-port)
(define image2-size (number->string (string-length image2-data)))

(define hdbc (make-connection-handle))
(display (connection-handle? hdbc)) (newline)

(define stmt0 "select count(0) from SCOTT.IMAGE where PIX_ID = ?")
(define stmt1 "insert into SCOTT.IMAGE (PIX_ID, PIX_NAME, PIX_TYPE,
BYTE_SIZE, PIX_WIDTH, PIX_HEIGHT, PIX_DATA) values (?, ?, ?, ?, ?, ?,
?)")

(if (db-login hdbc dsn uid pwd)
  (begin
    (display "\ndatabase login succeed !\n")
    (display (db-dbms hdbc)) (newline)
    (display (db-std-timestamp-format hdbc)) (newline)
    (display (db-max-long-data-size hdbc 2000000)) (newline)

    ; bind the query and insert statement
    (define hquery  (bind-statement hdbc stmt0))
    (define hinsert (bind-image-statement hdbc stmt1))

    (if (and
       (statement-handle? hquery)
       (statement-handle? hinsert)
     )
     (begin
       (if (not (query-exist hdbc hquery image1-id))
         (begin
          (display (string-append "insert image " image1-file "\n"))
          (if (and
            (db-stmt-param-assign hdbc hinsert 0 image1-id)
            (db-stmt-param-assign hdbc hinsert 1 image1-name)
            (db-stmt-param-assign hdbc hinsert 2 image1-type)
            (db-stmt-param-assign hdbc hinsert 3 image1-size)
            (db-stmt-param-assign hdbc hinsert 4 image1-width)
            (db-stmt-param-assign hdbc hinsert 5 image1-height)
            (db-stmt-param-assign hdbc hinsert 6 image1-data)
           )
           (if (execute-statement hdbc hinsert)
            (db-commit hdbc)
            (display (db-get-error-str hdbc))
           )
           (display (db-get-error-str hdbc))
          )
         )
      )

      (if (not (query-exist hdbc hquery image2-id))
         (begin
          (display (string-append "insert image " image2-file "\n"))
          (if (and
            (db-stmt-param-assign hdbc hinsert 0 image2-id)
            (db-stmt-param-assign hdbc hinsert 1 image2-name)
            (db-stmt-param-assign hdbc hinsert 2 image2-type)
            (db-stmt-param-assign hdbc hinsert 3 image2-size)
            (db-stmt-param-assign hdbc hinsert 4 image2-width)
            (db-stmt-param-assign hdbc hinsert 5 image2-height)
            (db-stmt-param-assign hdbc hinsert 6 image2-data)
           )
           (if (execute-statement hdbc hinsert)
            (db-commit hdbc)
            (display (db-get-error-str hdbc))
           )
           (display (db-get-error-str hdbc))
          )
         )
      )
     )
    )

    (if (not (db-logout hdbc))
     (display (db-get-error-str hdbc))
    )
  )
  (display (db-get-error-str hdbc))
)
```

### 5.8.8 Retrieving an Image from a Database

This sample shows how to Retrieve an image from a Database. It uses both Static and Dynamic SQL functions. See **“Static SQL Functions” on page 120** and **“Dynamic SQL Functions” on page 137** for more details.

```
;demo-image-select.monk

; load Monk database extension
(load "demo-init.monk")
(load "demo-common.monk")

(define (get-image hdbc hstmt)
   (do (
         (result (db-stmt-fetch hdbc hstmt) (db-stmt-fetch hdbc
hstmt))
         (first_name "")
         (file_type "")
         (file_name "")
         (width "")
         (height "")
         (output_port '())
       )
       ((boolean? result) result)
      (set! first_name (vector-ref result 0))
      (set! file_type (strip-trailing-whitespace (vector-ref result
1)))
      (set! width (strip-trailing-whitespace (vector-ref result 2)))
      (set! height (strip-trailing-whitespace (vector-ref result 3)))
      (cond
         ((string=? file_type "JPEG") (set! file_name (string-append
first_name ".jpg")))
         ((string=? file_type "GIF") (set! file_name (string-append
first_name ".gif")))
        ((string=? file_type "BITMAP") (set! file_name (string-append
first_name ".bmp")))
         ((string=? file_type "TIFF") (set! file_name (string-append
first_name ".tif")))
         (else (set! file_name (string-append first_name ".raw")))
      )
      (if (file-exists? file_name)
         (file-delete file_name)
      )
      (display (string-append "picture name = " file_name "\n"))
      (display (string-append "picture size = " width " x " height
"\n\n"))
      (set! output_port (open-output-file file_name))
      (display (vector-ref result 4) output_port)
      (close-port output_port)
   )
)

(define (execute-statement hdbc hstmt)
   (let ((col-count (db-stmt-column-count hdbc hstmt)) (row-count 0))
      (if (db-stmt-execute hdbc hstmt)
         (begin
            (if (> col-count 0)
               (if (not (get-image hdbc hstmt))
                  (display (db-get-error-str hdbc))
               )
            )
            (set! row-count (db-stmt-row-count hdbc hstmt))
            (if (boolean? row-count)
               (display (db-get-error-str hdbc))
              (display (string-append "number of image retrieved = "
(number->string row-count) "\n"))
            )
            (newline)
            #t
         )
         #f
      )
   )
)

(define hdbc (make-connection-handle))
(display (connection-handle? hdbc)) (newline)

(define stmt "select PIX_NAME, PIX_TYPE, PIX_WIDTH, PIX_HEIGHT,
PIX_DATA from SCOTT.IMAGE where PIX_ID = ?")

(if (db-login hdbc dsn uid pwd)
   (begin
      (display "\ndatabase login succeed !\n")
      (display (db-dbms hdbc)) (newline)
      (display (db-std-timestamp-format hdbc)) (newline)
      (display (db-max-long-data-size hdbc 2000000)) (newline)

      ; bind the select statement
      (define hselect (bind-binary-statement hdbc stmt))

      ; execute the dynamic statement
      (display "select IMAGE table\n")
      (if (statement-handle? hselect)
         (begin
            (if (db-stmt-param-assign hdbc hselect 0 "7100")
               (if (not (execute-statement hdbc hselect))
                  (display (db-get-error-str hdbc))
               )
               (display (db-get-error-str hdbc))
            )
            (if (db-stmt-param-assign hdbc hselect 0 "7200")
               (if (not (execute-statement hdbc hselect))
                  (display (db-get-error-str hdbc))
               )
               (display (db-get-error-str hdbc))
            )
         )
      )

      (if (not (db-logout hdbc))
         (display (db-get-error-str hdbc))
      )
   )
   (display (db-get-error-str hdbc))
)
```

### 5.8.9 Common Supporting Routines

This sample script displays and defines values and parameters for stored procedures. The routines contained in this script are used by many of the Monk samples in this chapter. For more details about functions used in this script, see **"Stored Procedure Functions" on page 150**

```
;demo-common.monk

;;
;; stored procedure auxiliary functions
;;

; display parameter properties of the stored procedure
(define (display-proc-parameter-property hdbc hstmt prm-count)
   (display "parameter count = ") (display prm-count) (newline)
   (do ((i 0 (+ i 1))) ((= i prm-count))
      (display "parameter ")
      (display (db-proc-param-name hdbc hstmt i))
      (display ": type = ")
      (display (db-proc-param-type hdbc hstmt i))
      (display ", io = ")
      (display (db-proc-param-io hdbc hstmt i))
      (newline)
   )
)

; display value of output parameters from stored procedure
(define (display-proc-parameter-output-value hdbc hstmt prm-count)
   (do ((i 0 (+ i 1))) ((= i prm-count))
      (if (not (equal? (db-proc-param-io hdbc hstmt i) "IN"))
         (begin
            (display "output parameter ")
            (display (db-proc-param-name hdbc hstmt i))
            (display " = ")
            (display (db-proc-param-value hdbc hstmt i))
            (newline)
         )
      )
   )
)

; display column properties of the return result set
(define (display-proc-column-property hdbc hstmt col-count)
   (display "column count = ") (display col-count) (newline)
   (do ((i 0 (+ i 1))) ((= i col-count))
      (display "column ")
      (display (db-proc-column-name hdbc hstmt i))
      (display ": type = ")
      (display (db-proc-column-type hdbc hstmt i))
      (newline)
   )
   (newline)
)

; display column value of the return result set of the stored
procedure
(define (display-proc-column-value hdbc hstmt col-count)
   (define (fetch-next)
      (let ((result (db-proc-fetch hdbc hstmt)))
         (if (boolean? result)
            result
            (begin (display result) (newline) (fetch-next))
         )
      )
   )
   (fetch-next)
   (newline)
)

; bind stored procedure and display parameter properties
(define (bind-procedure hdbc proc)
   (let ((hstmt (db-proc-bind hdbc proc)))
      (if (statement-handle? hstmt)
         (begin
            (display (string-append "bind stored procedure : " proc
"\n"))
            (define prm-count (db-proc-param-count hdbc hstmt))
            (display-proc-parameter-property hdbc hstmt prm-count)
            (newline)
            (if (db-proc-return-exist hdbc hstmt)
               (begin
                  (display "return: type = ")
                  (display (db-proc-return-type  hdbc hstmt))
                  (newline)
               )
            )
            (newline)
         )
          (display (db-get-error-str hdbc))
      )
      hstmt
   )
)



;;
;; dynamic statement auxiliary functions
;;

; display parameter properties of the SQL statement
(define (display-stmt-parameter-property hdbc hstmt prm-count)
   (display "parameter count = ") (display prm-count) (newline)
   (do ((i 0 (+ i 1))) ((= i prm-count))
      (display "parameter #")
      (display i)
      (display ": type = ")
      (display (db-stmt-param-type hdbc hstmt i))
      (newline)
   )
   (newline)
)

; display column properties of the SQL statement
(define (display-stmt-column-property hdbc hstmt col-count)
   (display "column count = ") (display col-count) (newline)
   (do ((i 0 (+ i 1))) ((= i col-count))
      (display "column ")
      (display (db-stmt-column-name hdbc hstmt i))
      (display ": type = ")
      (display (db-stmt-column-type hdbc hstmt i))
      (newline)
   )
   (newline)
)
```

```
; display column value of the return result set of the SQL statement
(define (display-stmt-column-value hdbc hstmt)
   (define (fetch-next)
      (let ((result (db-stmt-fetch hdbc hstmt)))
         (if (boolean? result)
            result
            (begin (display result) (newline) (fetch-next))
         )
      )
   )
   (fetch-next)
   (newline)
)

; display row count affected by the execution of the SQL statement
(define (display-stmt-row-count hdbc hstmt)
   (let ((row-count (db-stmt-row-count hdbc hstmt)))
      (cond
         ((= row-count 0) (display "\n(no row affected)\n"))
         ((= row-count 1) (display "\n(1 row affected)\n"))
         (else (display (string-append "\n(" (number->string row-
count) " rows affected)\n")))
      )
   )
)

; bind dynamic statement and display paramters and column properties
(define (bind-statement hdbc stmt)
   (let ((hstmt (db-stmt-bind hdbc stmt)))
      (display (string-append "\nDynamic statement : " stmt "\n"))
      (if (statement-handle? hstmt)
         (begin
            (define prm-count (db-stmt-param-count hdbc hstmt))
            (display-stmt-parameter-property hdbc hstmt prm-count)

            (define col-count (db-stmt-column-count hdbc hstmt))
            (display-stmt-column-property hdbc hstmt col-count)
          )
          (display (db-get-error-str hdbc))
      )
      hstmt
   )
)

; bind dynamic statement to input/output raw binary data
(define (bind-binary-statement hdbc stmt)
   (let ((hstmt (db-stmt-bind-binary hdbc stmt)))
      (display (string-append "\nDynamic statement : " stmt "\n"))
      (if (statement-handle? hstmt)
         (begin
            (define prm-count (db-stmt-param-count hdbc hstmt))
            (display-stmt-parameter-property hdbc hstmt prm-count)

            (define col-count (db-stmt-column-count hdbc hstmt))
            (display-stmt-column-property hdbc hstmt col-count)
          )
          (display (db-get-error-str hdbc))
      )
      hstmt
   )
)
```

# Index