Documentation Library

DataHub® ODBC Scripting

Version 7.3

Cogent Real-Time Systems, Inc.

August 15, 2012

DataHub® ODBC Scripting: Version 7.3

A user's guide to ODBC (Open DataBase Connectivity) scripting for the Cogent DataHub.

Published August 15, 2012
Cogent Real-Time Systems, Inc.
162 Guelph Street, Suite 253
Georgetown, Ontario
Canada, L7G 5X7

Toll Free: 1 (888) 628-2028
Tel: 1 (905) 702-7851
Fax: 1 (905) 702-7850

Information Email: info@cogent.ca
Tech Support Email: support@cogent.ca
Web Site: www.cogent.ca

Revision History

Revision 7.3-1 September 2007
Improved tutorials and added explanation of tutorial code.
Revision 6.2-1 October 2005
Initial release of documentation.

Copyright, trademark, and software license information.

Copyright Notice

Trademark Notice

END-USER LICENSE AGREEMENT FOR COGENT SOFTWARE

IMPORTANT - READ CAREFULLY: This End-User License Agreement ("EULA") is a legal agreement between you (either an individual or a single entity) and Cogent Real-Time Systems Inc. ("Cogent") of 162 Guelph Street, Suite 253, Georgetown, Ontario, L7G 5X7, Canada (Tel: 905-702-7851, Fax: 905-702-7850), from whom you acquired the Cogent software product(s) ("SOFTWARE PRODUCT" or "SOFTWARE"), either directly from Cogent or through one of Cogent's authorized resellers.
The SOFTWARE PRODUCT includes computer software, any associated media, any printed materials, and any "online" or electronic documentation. By installing, copying or otherwise using the SOFTWARE PRODUCT, you agree to be bound by the terms of this EULA. If you do not agree with the terms of this EULA, Cogent is unwilling to license the SOFTWARE PRODUCT to you. In such event, you may not use or copy the SOFTWARE PRODUCT, and you should promptly contact Cogent for instructions on return of the unused product(s) for a refund.

SOFTWARE PRODUCT LICENSE

The SOFTWARE PRODUCT is protected by copyright laws and copyright treaties, as well as other intellectual property laws and treaties. The SOFTWARE PRODUCT is licensed, not sold.

1. **EVALUATION USE:** This software is distributed as "Free for Evaluation", and with a per-use royalty for Commercial Use, where "Free for Evaluation" means to evaluate Cogent's software and to do exploratory development and "proof of concept" prototyping of software applications, and where "Free for Evaluation" specifically excludes without limitation:
    i. use of the SOFTWARE PRODUCT in a business setting or in support of a business activity,
    ii. development of a system to be used for commercial gain, whether to be sold or to be used within a company, partnership, organization or entity that transacts commercial business,
    iii. the use of the SOFTWARE PRODUCT in a commercial business for any reason other than exploratory development and "proof of concept" prototyping, even if the SOFTWARE PRODUCT is not incorporated into an application or product to be sold,
    iv. the use of the SOFTWARE PRODUCT to enable the use of another application that was developed with the SOFTWARE PRODUCT,
    v. inclusion of the SOFTWARE PRODUCT in a collection of software, whether that collection is sold, given away, or made part of a larger collection.
    vi. inclusion of the SOFTWARE PRODUCT in another product, whether or not that other product is sold, given away, or made part of a larger product.

2. **COMMERCIAL USE:** COMMERCIAL USE is any use that is not specifically defined in this license as EVALUATION USE.
3. **GRANT OF LICENSE:** This EULA covers both COMMERCIAL and EVALUATION USE of the SOFTWARE PRODUCT. Either clause (A) or (B) of this section will apply to you, depending on your actual use of the SOFTWARE PRODUCT. If you have not purchased a license of the SOFTWARE PRODUCT from Cogent or one of Cogent's authorized resellers, then you may not use the product for COMMERCIAL USE.
    A. **GRANT OF LICENSE (EVALUATION USE):** This EULA grants you the following non-exclusive rights when used for EVALUATION purposes:

Software: You may use the SOFTWARE PRODUCT on any number of computers, either stand-alone, or on a network, so long as every use of the SOFTWARE PRODUCT is for EVALUATION USE. You may reproduce the SOFTWARE PRODUCT, but only as reasonably required to install and use it in accordance with this LICENSE or to follow your normal back-up practices.

Subject to the license expressly granted above, you obtain no right, title or interest in or to the SOFTWARE PRODUCT or related documentation, including but not limited to any copyright, patent, trade secret or other proprietary rights therein. All whole or partial copies of the SOFTWARE PRODUCT remain property of Cogent and will be considered part of the SOFTWARE PRODUCT for the purpose of this EULA.

Unless expressly permitted under this EULA or otherwise by Cogent, you will not:

i. use, reproduce, modify, adapt, translate or otherwise transmit the SOFTWARE PRODUCT or related components, in whole or in part;

ii. rent, lease, license, transfer or otherwise provide access to the SOFTWARE PRODUCT or related components;

iii. alter, remove or cover proprietary notices in or on the SOFTWARE PRODUCT, related documentation or storage media;

iv. export the SOFTWARE PRODUCT from the country in which it was provided to you by Cogent or its authorized reseller;

v. use a multi-processor version of the SOFTWARE PRODUCT in a network larger than that for which you have paid the corresponding multi-processor fees;

vi. decompile, disassemble or otherwise attempt or assist others to reverse engineer the SOFTWARE PRODUCT;

vii. circumvent, disable or otherwise render ineffective any demonstration time-outs, locks on functionality or any other restrictions on use in the SOFTWARE PRODUCT;

viii. circumvent, disable or otherwise render ineffective any license verification mechanisms used by the SOFTWARE PRODUCT;

ix. use the SOFTWARE PRODUCT in any application that is intended to create or could, in the event of malfunction or failure, cause personal injury or property damage; or

x. make use of the SOFTWARE PRODUCT for commercial gain, whether directly, indirectly or incidentally.

B. **GRANT OF LICENSE (COMMERCIAL USE):** This EULA grants you the following non-exclusive rights when used for COMMERCIAL purposes:

Software: You may use the SOFTWARE PRODUCT on one computer, or if the SOFTWARE PRODUCT is a multi-processor version - on one node of a network, either: (i) as a development systems for the purpose of creating value-added software applications in accordance with related Cogent documentation; or (ii) as a single run-time copy for use as an integral part of such an application. This includes reproduction and configuration of the SOFTWARE PRODUCT, but only as reasonably required to install and use it in association with your licensed processor or to follow your normal back-up practices.

Storage/Network Use: You may also store or install a copy of the SOFTWARE PRODUCT on one computer to allow your other computers to use the SOFTWARE PRODUCT over an internal network, and distribute the SOFTWARE PRODUCT to your other computers over an internal network. However, you must acquire and dedicate a license for the SOFTWARE PRODUCT for each computer on which the SOFTWARE PRODUCT is used or to which it is distributed. A license for the SOFTWARE PRODUCT may not be shared or used concurrently on different computers.

Subject to the license expressly granted above, you obtain no right, title or interest in or to the SOFTWARE PRODUCT or related documentation, including but not limited to any copyright, patent, trade secret or other proprietary rights therein. All whole or partial copies of the SOFTWARE PRODUCT remain property of Cogent and will be considered part of the SOFTWARE PRODUCT for the purpose of this EULA.

Unless expressly permitted under this EULA or otherwise by Cogent, you will not:

i. use, reproduce, modify, adapt, translate or otherwise transmit the SOFTWARE PRODUCT or related components, in whole or in part;

ii. rent, lease, license, transfer or otherwise provide access to the SOFTWARE PRODUCT or related components;

iii. alter, remove or cover proprietary notices in or on the SOFTWARE PRODUCT, related documentation or storage media;

iv. export the SOFTWARE PRODUCT from the country in which it was provided to you by Cogent or its authorized reseller;

v. use a multi-processor version of the SOFTWARE PRODUCT in a network larger than that for which you have paid the corresponding multi-processor fees;

vi. decompile, disassemble or otherwise attempt or assist others to reverse engineer the SOFTWARE PRODUCT;

vii. circumvent, disable or otherwise render ineffective any demonstration time-outs, locks on functionality or any other restrictions on use in the SOFTWARE PRODUCT;

viii. circumvent, disable or otherwise render ineffective any license verification mechanisms used by the SOFTWARE PRODUCT, or

ix. use the SOFTWARE PRODUCT in any application that is intended to create or could, in the event of malfunction or failure, cause personal injury or property damage.

4. **WARRANTY:** Cogent cannot warrant that the SOFTWARE PRODUCT will function in accordance with related documentation in every combination of hardware platform, software environment and SOFTWARE PRODUCT configuration. You acknowledge that software bugs are likely to be identified when the SOFTWARE PRODUCT is used in your particular application. You therefore accept the responsibility of satisfying yourself that the SOFTWARE PRODUCT is suitable for your intended use. This includes conducting exhaustive testing of your application prior to its initial release and prior to the release of any related hardware or software modifications or enhancements.

   Subject to documentation errors, Cogent warrants to you for a period of ninety (90) days from acceptance of this EULA (as provided above) that the SOFTWARE PRODUCT as delivered by Cogent is capable of performing the functions described in related Cogent user documentation when used on appropriate hardware. Cogent also warrants that any enclosed disk(s) will be free from defects in material and workmanship under normal use for a period of ninety (90) days from acceptance of this EULA. Cogent is not responsible for disk defects that result from accident or abuse. Your sole remedy for any breach of warranty will be either: i) terminate this EULA and receive a refund of any amount paid to Cogent for the SOFTWARE PRODUCT, or ii) to receive a replacement disk.

5. **LIMITATIONS:** Except as expressly warranted above, the SOFTWARE PRODUCT, any related documentation and disks are provided "as is" without other warranties or conditions of any kind, including but not limited to implied warranties of merchantability, fitness for a particular purpose and non-infringement. You assume the entire risk as to the results and performance of the SOFTWARE PRODUCT. Nothing stated in this EULA will imply that the operation of the SOFTWARE PRODUCT will be uninterrupted or error free or that any errors will be corrected. Other written or oral statements by Cogent, its representatives or others do not constitute warranties or conditions of Cogent.

   In no event will Cogent (or its officers, employees, suppliers, distributors, or licensors: collectively "Its Representatives") be liable to you for any indirect, incidental, special or consequential damages whatsoever, including but not limited to loss of revenue, lost or damaged data or other commercial or economic loss, arising out of any breach of this EULA, any use or inability to use the SOFTWARE PRODUCT or any claim made by a third party, even if Cogent (or Its Representatives) have been advised of the possibility of such damage or claim. In no event will the aggregate liability of Cogent (or that of Its Representatives) for any damages or claim, whether in contract, tort or otherwise, exceed the amount paid by you for the SOFTWARE PRODUCT.

   These limitations shall apply whether or not the alleged breach or default is a breach of a fundamental condition or term, or a fundamental breach. Some jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, or certain limitations of implied warranties. Therefore the above limitation may not apply to you.

6. **DESCRIPTION OF OTHER RIGHTS AND LIMITATIONS:**

   Separation of Components. The SOFTWARE PRODUCT is licensed as a single product. Its component parts may not be separated for use on more than one computer.

Termination. Without prejudice to any other rights, Cogent may terminate this EULA if you fail to comply with the terms and conditions of this EULA. In such an event, you must destroy all copies of the SOFTWARE PRODUCT and all of its component parts.

7. **UPGRADES:** If the SOFTWARE PRODUCT is an upgrade from another product, whether from Cogent or another supplier, you may use or transfer the SOFTWARE PRODUCT only in conjunction with that upgrade product, unless you destroy the upgraded product. If the SOFTWARE PRODUCT is an upgrade of a Cogent product, you now may use that upgraded product only in accordance with this EULA. If the SOFTWARE PRODUCT is an upgrade of a component of a package of software programs which you licensed as a single product, the SOFTWARE PRODUCT may be used and transferred only as part of that single product package and may not be separated for use on more than one computer.
8. **COPYRIGHT:** All title and copyrights in and to the SOFTWARE PRODUCT (including but not limited to any images, photographs, animations, video, audio, music, text and ’applets", incorporated into the SOFTWARE PRODUCT), any accompanying printed material, and any copies of the SOFTWARE PRODUCT, are owned by Cogent or its suppliers. You may not copy the printed materials accompanying the SOFTWARE PRODUCT. All rights not specifically granted under this EULA are reserved by Cogent.
9. **PRODUCT SUPPORT:** Cogent has no obligation under this EULA to provide maintenance, support or training.
10. **RESTRICTED RIGHTS:** Use, duplication, or disclosure by the U.S. Government is subject to restrictions as provided in DFARS 227.7202-1(a) and 227.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii) (OCT 1988), FAR 12.212(a)(1995), FAR 52.227-19, or FAR 52.227-14 (ALT III), as appropriate. Manufacturer is Cogent Real-Time Systems Inc. 162 Guelph Street, Suite 253, Georgetown, Ontario, L7G 5X7, Canada.
11. **GOVERNING LAW:** This Software License Agreement is governed by the laws of the Province of Ontario, Canada. You irrevocably attorn to the jurisdiction of the courts of the Province of Ontario and agree to commence any litigation that may arise hereunder in the courts located in the Judicial District of Peel, Province of Ontario.

Table of Contents

Chapter 1. Introduction

1.1. Overview

The Cogent DataHub has built-in scripting capabilities (see DataHub Scripting), which among other things let you connect the DataHub to any ODBC-compliant database. This guide assumes a basic understanding of DataHub scripting, and provides the specific information you will need to script connections between the DataHub and an ODBC database.

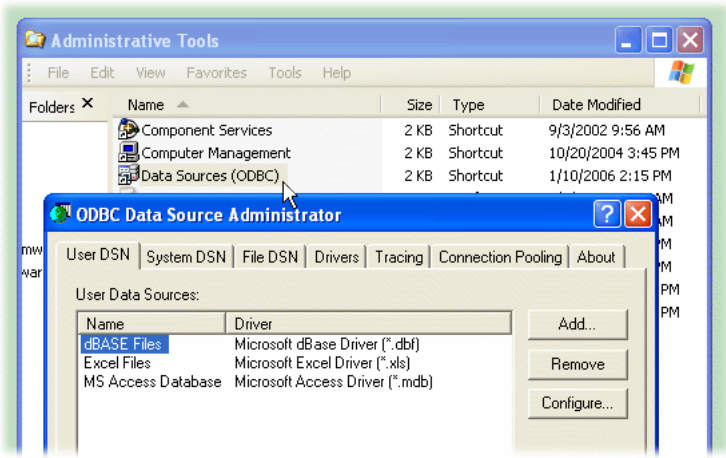
DataHub ODBC scripting uses wrapped MSDN functions. Therefore, this documentation often links to and refers to the MSDN documentation.

The tutorials in this manual use SQL commands to query the database. The syntax for these commands may vary slightly from one ODBC database to another. If a given tutorial doesn't work right away, check the syntax of the SQL commands in the tutorial against the syntax that your database uses.

1.2. Setting up a DSN (Data Source Name)

To connect an ODBC-compliant database to the DataHub, you will need to ensure that you have specified a *DSN* (Data Souce Name) for the database. Here's how:

1. From the Windows Start menu, choose Control Panel, then Administrative Tools, and then Data Sources (ODBC) to open the ODBC Data Source Administrator window. This is what it looks like in Windows XP:



2. Select the User DSN or System DSN tab, depending on how you plan to access your database.

   A user DSN is only available to the current user account, while a system DSN is available to any user account on the computer.

3. Now you can add a new database or configure an existing one.

   Add a new database

   1. Click the Add button. The Create New Data Source window will open, displaying a list of data source drivers.

2. Select the data source driver that corresponds to your ODBC database. A data source setup window will open. Each data source setup window is different, but you should be able to find the appropriate entry fields easily enough.
3. Enter the data source name and select the database.
4. Enter any other required or optional information such as login name, password, etc. What entries need to be made and where they are entered depends on the particular data source setup window you are using.
5. Click OK to return to the ODBC Data Source Administrator window. You should be able to see the new database and driver listed. If you need to make any changes, you can configure an exiting database, as explained below.

Configure an existing database

1. Select a data source name and click the Configure... button. This takes you to the data source setup window (explained above) where you can make changes to the configuration.
2. Make your changes and click OK to return to the ODBC Data Source Administrator window. Any time you need to make a change, you can go to this window.

4. When you are satisfied everything is correct, click the OK button to exit the ODBC Data Source Administrator.

1.3. Working with MS Access

The Microsoft Access database program is a handy tool for MS Office users, but it is not completely ODBC compliant, nor is it a database server. Its design prevents simultaneous updates from outside data sources while the Access program is running, but it can still be used with the DataHub to collect and store data from real-time systems.

File-based Data Access

MS Access is not a database server like MS SQL Server, MySQL, Oracle, and others. Instead, it accesses a data file (`.mdb` file), reading from and writing data to that file. Other programs like the DataHub can also access the file, but *not simultaneously* with Access. You can use the DataHub to modify the data file, but any time you open the file in Access, all programs including the DataHub are blocked from using it until you close the file in Access.

What this means is that if you are using the DataHub to interface to a real-time control or financial system and you want mission critical data stored in an ODBC-compliant database, you probably don't want to be using MS Access. However, it could be useful for storing and viewing logged data. And for some it might be a convenient way to start investigating the possibilities of ODBC scripting with the DataHub.

Queries on Primary Keys

MS Access does not support the ODBC function `SQLPrimaryKeys`, which means you cannot programatically discover the primary keys of an Access database. Thus you will need to identify the primary keys of the tables in the code itself. You will notice in our tutorials that we do just that. Since data table design doesn't change frequently, this should not prove to be a problem in most cases.

Chapter 2. Tutorials

2.1. Tutorial 1: Writing new rows to a table, based on a trigger - Multi-Threaded Version

This script creates and inserts a new row into a database whenever a trigger point changes value. The data that gets inserted into the row is an ID for the entry (the primary key), the name of a specified point, its value, and a timestamp of the change. The script uses the multi-threaded feature Store and Forward to store data in memory and/or on disk if the database is disconnected or too busy, and then log that data in time-sequential order when the database is available again.

The tutorials in this manual use SQL commands to query the database. The syntax for these commands may vary slightly from one ODBC database to another. If a given tutorial doesn't work right away, check the syntax of the SQL commands used here against the syntax that your database uses.

Getting Started

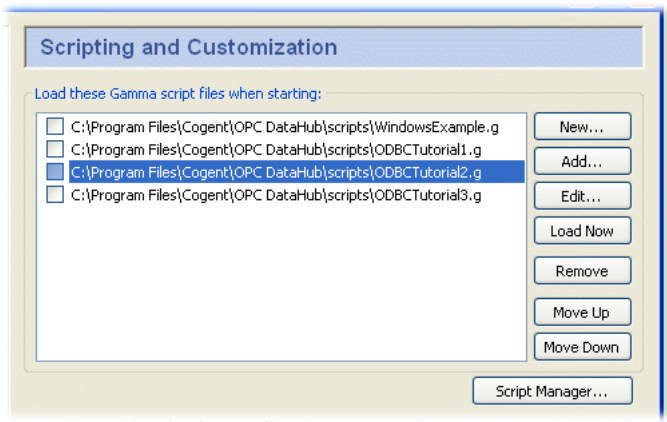
To run this code or the other tutorials in this manual, you will need to do the following:

1. Set up a DSN (Data Source Name) called "`DataHubThreadedTest`" and point it to an empty database on your database server.
2. Create a table in the database named "`datatable`" that contains at least four columns with names, data types, and other attributes exactly as specified here:

Column name	Data type	Other attributes
`ptid`	integer	identity, non-null, counter
`ptname`	text string	null
`ptvalue`	real	non-null
`pttime`	datetime	null

   Any other columns in this table must be allowed to take on a null value.
3. Start DataSim.
4. Find the tutorial script `ODBCTutorial1.g` on your system, and run it.

You can access DataHub scripts and scripting capabilities by pressing the Scripting button in the Properties window, to display the Scripting and Customization screen. The upper half of the screen shows the Gamma files currently configured in the DataHub:



The Open button opens a file selector for you to add an existing script to the list. Scripts are normally kept in the DataHub's `scripts` subdirectory, `C:\Program Files\Cogent\DataHub\scripts\`*`myscript`*`.g`.
The Edit button opens the selected script in the Script Editor for editing.
You can view error message and printed output from a script in the Script Log. To open the Script Log, right click on the DataHub icon in the system tray, and select View Script Log.
For complete information about DataHub scripting, please refer to the DataHub Scripting Manual

5. Check the database table to see the results. Once you have it working, you can modify the code as explained below.

The Code: `ODBCTutorial1.g`

```
/*
 * This script demonstrates the use of the threaded ODBC interface to insert
 * data from the DataSim program into a database based on a timer or an event.
 */

require ("Application");
require ("ODBCThreadSupport");
require ("Time");
require ("Quality");

class ODBCTutorial1 Application
{
    DSN = "MySQLLocal";      // The DSN name to use for the database connection
    username = "test";       // The user name for connecting to the database
    password = "test";       // The password for connecting to the database
    tablename = "test";      // The name of the database table
    cachefile = "c:/tmp/testcache.txt";     // Base name for the disk cache file
    tableclass;
    thread;
}

/* This method will be called every time the connection is established to the database.
 * If there is something we only want to perform on the first connection, we can test
 * is_first_connect to perform the code only once.
 */
method ODBCTutorial1.onConnect()
{
    princ ("Connection succeeded\n");
    if (.thread.is_first_connect)
    {
        // Start the sequence defined by the AddInitStage calls in the constructor
```

```
        .thread.BeginAsyncInit();
    }
}

/* If we get a connection attempt failure, or the connection fails after having been
 * connected, this method is called.
 */
method ODBCTutorial1.onConnectFail()
{
    princ ("Connection closed: ", SQLResult.Description, "\n");
}

/* Map the table in the set of table definitions that matches the name in .tablename
 * into a Gamma class.  This lets us easily convert between class instances and rows
 * in the table.
 */
method ODBCTutorial1.mapTable(name, tabledefinitions)
{
    .tableclass = .thread.ClassFromTable(name, tabledefinitions);
}

/* Set up the timer or event handler functions to write to the table. */
method ODBCTutorial1.startLogging()
{
     /* You can modify and/or add similar timers or event handlers for
      * each data point that you want to log.  Please refer to the "Methods
      * and Functions from Application.g" section of the documentaton
      * for more details about the timer and event handler funtions.
      * http://www.cogentdatahub.com/Docs/dhs-reference-applicationg.html
      */
     // Log a new row of data every 3 seconds.
    .TimerEvery(3, '(@self).writeData(#$DataSim:Sine));

     // Log a new row of data at 20 seconds past each minute of each hour, etc.
    .TimerAt(nil, nil, nil, nil, nil, 20, '(@self).writeData(#$DataSim:Triangle));

    // Log a new row of data for the point DataSim:Square when it changes.
    .OnChange(#$DataSim:Square, '(@self).writeData(this));

    // Log a new row of data for the point DataSim:Sine when DataSim:Square changes.
    .OnChange(#$DataSim:Square, '(@self).writeData(#$DataSim:Sine));
}

method ODBCTutorial1.writeData(pointsymbol)
{
    local       row = new (.tableclass);
    local       pttime, ptltime;
    local       timestring;

    // Generate a timestamp in database-independent format to the millisecond.
    // Many databases strip the milliseconds from a timestamp, but it is harmless
    // to provide them in case the database can store them.
    pttime = WindowsTimeToUnixTime(PointMetadata(pointsymbol).timestamp);
    ptltime = localtime(pttime);
    timestring = format("{ts '%04d-%02d-%02d %02d:%02d:%02d.%03d'}",
        ptltime.year+1900, ptltime.mon+1, ptltime.mday, ptltime.hour, ptltime.min, ptltime.sec,
        (pttime % 1) * 1000);

    // Fill the row.  Since we mapped the table into a Gamma class, we can access
    // the columns in the row as member variables of the mapped class.
    row.ptname = string(pointsymbol);
    row.ptvalue = eval(pointsymbol);
    row.pttime = timestring;
    // Perform the insertion.  In this case we are providing no callback on completion.
    .thread.Insert(row, nil);
}

/* Write the 'main line' of the program here. */
```

```
method ODBCTutorial1.constructor ()
{
    // Create and configure the database connection object
    .thread = new ODBCThread();
    .thread.Configure(.DSN, .username, .password, STORE_AND_FORWARD, .cachefile, 0);

    // Use this to delete the table on the first connection after the script starts.
    // BE CAREFUL - re-running the script will start over and delete the table again.
    // .thread.AddInitStage(format("drop table %s", .tablename), nil, t);

    // Use this to create the table if it does not exist.  Note: this might not work for all databases.
    // When in doubt, create the table manually.  The 't' in the onFail argument says to ignore errors
    // and continue with the next stage.
    // .thread.AddInitStage(format("create table %s (ptid int auto_increment primary key, ptname varchar(64),
    //                             ptvalue double, pttime datetime )", .tablename), nil, t);

    // Query the table and map it to a class for each insertion.  We want to run an asynchronous event
    // within the asynchronous initialization stage, so to do that we specify the special method
    // cbInitStage as the callback function of our asynchronous event (GetTableInfo).  We deal with
    // the return from the GetTableInfo in the onSuccess argument of the init stage.
    .thread.AddInitStage('(@.thread).GetTableInfo("", "", (@.tablename), "TABLE,VIEW",
                                                  '(@.thread).cbInitStage()),
                         '(@self).mapTable(@.tablename, SQLTables), nil);

    // Do not start writing data to the table until we have successfully created and mapped
    // the table to a class.  If we wanted to start writing data immediately, then we would
    // create the table class beforehand instead of querying the database for the table
    // definition.  Then, even if the database were unavailable we could still cache to the
    // local disk until the database was ready.
    .thread.AddInitStage(nil, '(@self).startLogging(), nil);

    // Set up the callback functions for various events from the database thread
    .thread.OnConnectionSucceeded = '(@self).onConnect();
    .thread.OnConnectionFailed = '(@self).onConnectFail();
    .thread.OnFileSystemError = 'princ("File System Error: ", SQLResult, "\n");
    .thread.OnODBCError = 'princ("ODBC Error: ", SQLResult, "\n");
    .thread.OnExecuteStored = nil;

    // Now that everything is configured, start the thread and begin connecting.  All of the
    // logic now will be driven through the onConnect callback and then through the init
    // stages.
    .thread.Start();

    // Create a menu item in the system tray that allows us to open a window to monitor
    // the performance of the ODBC thread.  The menu strings can be edited as desired.
    .AddCustomSubMenu("ODBC Thread Demo");
    .AddCustomMenuItem("Monitor Performance",
                       '(@.thread).CreateMonitorWindow((@self), "ODBC Demo Monitor"));

    // If we want to open the performance monitor window when the script starts, do it here.
    .thread.CreateMonitorWindow(self, "ODBC Demo Monitor");
}

/* Any code to be run when the program gets shut down. */
method ODBCTutorial1.destructor ()
{
 if (instance_p(.thread))
  destroy(.thread);
}

/* Start the program by instantiating the class. */
ApplicationSingleton (ODBCTutorial1);
```

Modifying the Code

You can modify the `startLogging` method to add your own points by replacing data domains (*`domain`*), point names (*`point`*) and/or times (*`day`*, *`month`*, *`year`*, *`hour`*, *`minute`*, *`second`*, etc.) like this:

```
 // Log a new row of data every # seconds.
.TimerEvery(seconds, '(@self).writeData(#$domain:point));

 // Log a new row of data at # seconds past each minute of each hour, etc.
.TimerAt(day, month, year, hour, minute, second, '(@self).writeData(#$domain:point));

// Log a new row of data for a point when it changes.
.OnChange(#$domain:point, '(@self).writeData(this));

// Log a new row of data for a point when a trigger point changes.
.OnChange(#$domain:point, '(@self).writeData(#$domain:point));
```

Please refer to the documentation for these methods of the `Application` class for more information: `TimerEvery`, `TimerAt`, and `OnChange`.

2.2. Tutorial 2: Writing new rows to a table, based on a trigger - Single-Threaded Version

This script creates and inserts a new row into a database whenever a trigger point changes value. The data that gets inserted into the row is an ID for the entry (the primary key), the value of a specified point, the timestamp of the change, and the name and quality of the point. The script also checks the connection to the database, and will attempt to reconnect every 5 seconds if the connection is lost.

The tutorials in this manual use SQL commands to query the database. The syntax for these commands may vary slightly from one ODBC database to another. If a given tutorial doesn't work right away, check the syntax of the SQL commands used here against the syntax that your database uses.

Getting Started

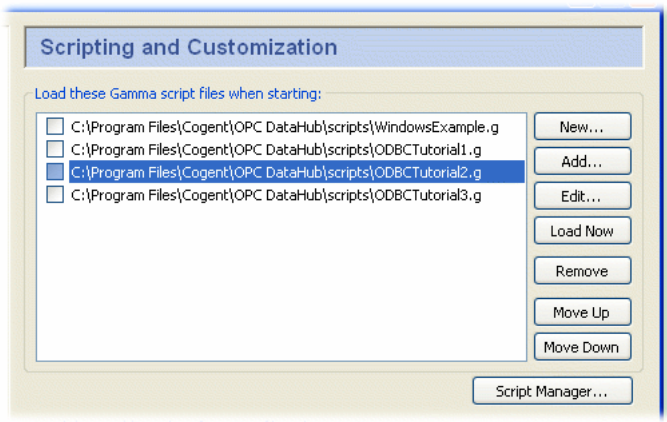
To run this code or the other tutorials in this manual, you will need to do the following:

1. Set up a DSN (Data Source Name) called "`DataHubTest`" and point it to an empty database on your database server.
2. Create a table in the database named "`datatable`" that contains at least five columns with names, data types, and other attributes exactly as specified here:

Column name	Data type	Other attributes
`ID`	integer	identity, non-null, counter
`PTVALUE`	real	non-null
`PTTIME`	datetime	null
`PTNAME`	text string	null
`PTQUALITY`	text string	null

   Any other columns in this table must be allowed to take on a null value.
3. Start DataSim.
4. Find the tutorial script `ODBCTutorial2.g` on your system, and run it.

You can access DataHub scripts and scripting capabilities by pressing the Scripting button in the Properties window, to display the Scripting and Customization screen. The upper half of the screen shows the Gamma files currently configured in the DataHub:



The Open button opens a file selector for you to add an existing script to the list. Scripts are normally kept in the DataHub's `scripts` subdirectory, `C:\Program Files\Cogent\DataHub\scripts\`*`myscript`*`.g`.
The Edit button opens the selected script in the Script Editor for editing.
You can view error message and printed output from a script in the Script Log. To open the Script Log, right click on the DataHub icon in the system tray, and select View Script Log.
For complete information about DataHub scripting, please refer to the DataHub Scripting Manual

5. Check the database table to see the results. Once you have it working, you can modify the code as explained below.

The Code: `ODBCTutorial2.g`

```
/* All user scripts should derive from the base "Application" class */

require ("Application");

/* Get the Gamma library functions and methods for ODBC and/or
 * Windows programming.  Uncomment either or both. */

//require ("WindowsSupport");
require ("ODBCSupport");
require ("Time");
require ("Quality");

/* Applications share the execution thread and the global name
 * space, so we create a class that contains all of the functions
 * and variables for the application.  This does two things:
 *   1) creates a private name space for the application, and
 *   2) allows you to re-load the application to create either
 *      a new unique instance or multiple instances without
 *      damaging an existing running instance.
 */

/*
 * This application assumes that the table specified by the "tablename"
 * member variable exists in the DSN specified by the "DSN" member
 * variable below.
 * The table consists of at least the following columns:
 *    ID - integer, identity, non-null, counter
 *    PTVALUE - real, non-null
 *    PTTIME - datetime, null
 *    PTNAME - text string, null
 *    PTQUALITY - text string, null
```

```
 * Any other columns in this table must be allowed to take on a
 * NULL value.
 */

class ODBCTutorial2 Application
{
    /* User-defined values, may be changed as needed. */
    DSN = "DataHubTest";
    user = "test";
    password = "test";
    tablename = "datatable";

    /* These values get defined by the program.*/
    conn;
    env;
    tableclass;
    is_connected;
    is_connecting;
}

/* Connect to the DSN and create a class that maps the table. */
method ODBCTutorial2.Connect ()
{
    princ ("Connecting to database\n");
    .is_connecting = t;

    protect
    {
        /* Create the ODBC environment and connection */
        if (!.env)
            .env = ODBC_AllocEnvironment();
        if (!.conn)
            .conn = .env.AllocConnection();

        /* Attempt the connection. */
        ret = .conn.Connect (.DSN, .user, .password);
        if (ret != SQL_SUCCESS && ret != SQL_SUCCESS_WITH_INFO)
            error (.conn.GetDiagRec());

        /* Create a class from the table */
        .tableclass = .conn.ClassFromTable (#DataEntry, nil, .tablename);

        /* Set the primary key.  This is redundant for MS-SQL and MYSQL since
           they can figure it out themselves, but Access requires it. */
        mykey = .conn.SetPrimaryKey (.tableclass, "ID");

        .is_connected = t;
    }
    unwind
    {
        .is_connecting = nil;
        if (.is_connected)
            princ ("    Connection successful\n");
        else
            princ ("    Connection failed\n");
    }
}

/* Disconnect from the database server */
method ODBCTutorial2.Disconnect ()
{
    if (.conn)
    {
        try
        {
            princ ("Disconnecting database\n");
            .conn.Disconnect();
            destroy(.conn);
```

```
        }
        catch
        {
            princ ("Disconnection failed: ", _last_error_, "\n");
        }
        .conn = nil;
    }
    .is_connected = nil;
}

method ODBCTutorial2.Reconnect()
{
    if (!.is_connected && !.is_connecting)
    {
        .Connect();
    }
}

/* Fill a database record with new information from a point change. */
method ODBCTutorial2.FillRecord (record, sym, newvalue)
{
    local   timestamp;

    timestamp = localtime (PointGetUnixTime(sym));
    timestamp = format ("%d-%02d-%02d %02d:%02d:%02d",
        timestamp.year + 1900, timestamp.mon + 1,
        timestamp.mday, timestamp.hour, timestamp.min,
        timestamp.sec);
    record.PTNAME = string (sym);
    record.PTVALUE = number (newvalue);
    record.PTTIME = timestamp;
    record.PTQUALITY = GetQualityName(PointMetadata(sym).quality);
    record;
}

/* Write a new record into the database based on a point change. */
method ODBCTutorial2.AddRecord (sym, newvalue)
{
    local    record = new (.tableclass);
    .FillRecord (record, sym, newvalue);
    try
    {
        .conn.Insert (record);
    }
    catch
    {
        princ ("Write failed.  Disconnecting.  Record was not written.\n");
        .Disconnect();
    }
    record;
}

/* The mainline.  Connect to the database and begin storing data from
   the DataHub into the database. */
method ODBCTutorial2.constructor ()
{
    local    ret;

    /* Start a timer that will reconnect to the database every 5 seconds
       if the connection has been lost. */
    .TimerEvery(5, '(@self).Reconnect());

    /* Try connecting now.  If this fails, the timer will try again later. */
    .Reconnect();

    /* Add a record when a point changes.  */
    .OnChange (#$DataSim:Square, '(@self).AddRecord (#$DataSim:Sine, $DataSim:Sine));
```

```
    /* Add more points like this:
     *   .OnChange (#$DataSim:Square, '(@self).AddRecord (#$MyDomain:MyPt, $MyDomian:MyPt));
     * Have the trigger point's value get written like this:
     *   .OnChange (#$DataSim:Square, '(@self).AddRecord (#$DataSim:Square, $DataSim:Square));
     */
}

/* Any code to be run when the program gets shut down. */
method ODBCTutorial2.destructor ()
{
    .Disconnect();
    if (.env)
        destroy(.env);
}

/* Start the program by instantiating the class.  If your
 * constructor code does not create a persistent reference to
 * the instance (self), then it will be destroyed by the
 * garbage collector soon after creation.  If you do not want
 * this to happen, assign the instance to a global variable, or
 * create a static data member in your class to which you assign
 * 'self' during the construction process.  ApplicationSingleton()
 * does this for you automatically. */
ApplicationSingleton (ODBCTutorial2);
```

Modifying the Code

There are several ways you can modify the code. These modifications are made in the `ODBCTutorial1.constructor` method, towards the end of the script.

- **Add more DataSim points** using this format:

  ```
  .OnChange (#$DataSim:Square, '(@self).AddRecord (#$DataSim:pointname, $DataSim:pointname));
  ```

  Where *`pointname`* is the name of a point in DataSim.

- **Change the trigger point** like this:

  ```
  .OnChange (#$DataSim:pointname, '(@self).AddRecord (#$DataSim:Sine, $DataSim:Sine));
  ```

  Where *`pointname`* is the name of a point in DataSim.

- **Add your own points** You can add your own points using this syntax:

  ```
  .OnChange (#$domain:pointname, '(@self).AddRecord (#$domain:pointname, $domain:pointname));
  ```

  where *`domain`* is the domain that the point is in, and *`pointname`* is the name of the point.

2.3. Tutorial 3: Updating existing rows, or writing new ones

This tutorial demonstrates how to find a particular row and update it, as well as write new rows, depending on the point.

This tutorial uses the same DSN, database, and table as Tutorial 2. If you haven't done Tutorial 2 yet, please review Getting Started in that section to see how to set up your system for this tutorial.

The complete code for this tutorial is shown below, and is included in your DataHub distribution, in the `scripts` subdirectory, such as `C:\Program Files\Cogent\DataHub\scripts\`*`myscript.g`*. Please refer to Accessing Scripts in the DataHub Scripting manual for details on how to load and run a script.

The Code: `ODBCTutorial3.g`

```
/* All user scripts should derive from the base "Application" class */

require ("Application");

/* Get the Gamma library functions and methods for ODBC and/or
 * Windows programming.  Uncomment either or both. */

//require ("WindowsSupport");
require ("ODBCSupport");
require ("Time");
require ("Quality");

/* Applications share the execution thread and the global name
 * space, so we create a class that contains all of the functions
 * and variables for the application.  This does two things:
 *   1) creates a private name space for the application, and
 *   2) allows you to re-load the application to create either
 *      a new unique instance or multiple instances without
 *      damaging an existing running instance.
 */

/*
 * This application assumes that the table specified by the "tablename"
 * member variable exists in the DSN specified by the "DSN" member
 * variable below.
 * The table consists of at least the following columns:
 *    ID - integer, identity, non-null, counter
 *    PTVALUE - real, non-null
 *    PTTIME - datetime, null
 *    PTNAME - text string, null
 *    PTQUALITY - text string, null
 * Any other columns in this table must be allowed to take on a
 * NULL value.
 */

class ODBCTutorial3 Application
{
    /* User-defined values, may be changed as needed. */
    DSN = "DataHubTest";
    user = "test";
    password = "test";
    tablename = "datatable";

    /* These values get defined by the program.*/
    conn;
    env;
    tableclass;
}

/* Connect to the DSN and create a class that maps the table. */
method ODBCTutorial3.Connect ()
{
    /* Create the ODBC environment and connection */
    .env = ODBC_AllocEnvironment();
    .conn = .env.AllocConnection();

    /* Attempt the connection. */
    ret = .conn.Connect (.DSN, .user, .password);
    if (ret != SQL_SUCCESS && ret != SQL_SUCCESS_WITH_INFO)
        error (.conn.GetDiagRec());
```

```
    /* Create a class from the table */
    .tableclass = .conn.ClassFromTable (#DataEntry, nil, .tablename);

    /* Set the primary key.  This is redundant for MS-SQL and MYSQL since
       they can figure it out themselves, but Access requires it. */
    mykey = .conn.SetPrimaryKey (.tableclass, "ID");
}

/* Fill a database record with new information from a point change */
method ODBCTutorial3.FillRecord (record, sym, newvalue)
{
    local    timestamp;

    timestamp = localtime (PointGetUnixTime(sym));
    timestamp = format ("%d-%02d-%02d %02d:%02d:%02d",
        timestamp.year + 1900, timestamp.mon + 1,
        timestamp.mday, timestamp.hour, timestamp.min,
        timestamp.sec);
    record.PTNAME = string (sym);
    record.PTVALUE = number (newvalue);
    record.PTTIME = timestamp;
    record.PTQUALITY = GetQualityName(PointMetadata(sym).quality);
    record;
}

/* Write a new record into the database based on a point change. */
method ODBCTutorial3.AddRecord (sym, newvalue)
{
    local    record = new DataEntry();
    .FillRecord (record, sym, newvalue);
    .conn.Insert (record);
    record;
}

/* Write a data point into a field of a record.  This is called
   from a DataHub point change event.  This method will replace
   an existing record that is cached with the point at startup.  If
   there was no existing row in the database, this will create one
   and then update it in subsequent calls. */
method ODBCTutorial3.UpdateRecord (sym, newvalue)
{
    local    record = getprop (sym, #dbrecord);
    if (!record)
    {
        record = .AddRecord (sym, newvalue);
        setprop (sym, #dbrecord, record);
    }
    else
    {
        .FillRecord (record, sym, newvalue);
        .conn.Update (record);
    }
}

/* Find an existing record in the database for this point.  If it
   exists, associate the record with the point. */
method ODBCTutorial3.GetExistingRecord (sym, klass)
{
    local    result = .conn.QueryToClass (klass, string
                                          ("SELECT * FROM ",
                                           klass.__table,
                                           " WHERE PTNAME = '", sym, "'"));
    if (array_p(result))
     setprop (sym, #dbrecord, result[0]);
}

/* Start updating the database whenever a point changes.  If the
   overwrite argument is non-nil or absent, then this method will
```

```
   cause an existing record in the database to be overwritten each
   time.  If overwrite is nil, every point change will create a
   new row in the database. */
method ODBCTutorial3.WatchPoint (sym, tableclass, overwrite?=t)
{
    /* Grab an existing record for this point if it exists */
    .GetExistingRecord (sym, tableclass);
    if (overwrite)
        .OnChange (sym, '(@self).UpdateRecord (this, value));
    else
        .OnChange (sym, '(@self).AddRecord (this, value));
}

/* The mainline.  Connect to the database and begin storing data from
   the DataHub into the database. */
method ODBCTutorial3.constructor ()
{
    local    ret;

    /* Connect to the DSN. */
    .Connect();

    /* Register points that we want to save.  The WatchPoint method takes
       an optional third arguement.  If it is nil, every point change will
       add a row to the table.  If it is absent or non-nil, then every point
       change overwrites the existing row in the table for that point. */
    .WatchPoint (#$DataSim:Square, .tableclass, nil);
    .WatchPoint (#$DataSim:Sine, .tableclass);
}

/* Any code to be run when the program gets shut down. */
method ODBCTutorial3.destructor ()
{
}

/* Start the program by instantiating the class.  If your
 * constructor code does not create a persistent reference to
 * the instance (self), then it will be destroyed by the
 * garbage collector soon after creation.  If you do not want
 * this to happen, assign the instance to a global variable, or
 * create a static data member in your class to which you assign
 * 'self' during the construction process.  ApplicationSingleton()
 * does this for you automatically. */
ApplicationSingleton (ODBCTutorial3);
```

Modifying the Code

There are several ways you can modify the code. These modifications are made in the `ODBCTutorial2.constructor` method, towards the end of the script.

- **Overwrite or add rows.** There are two ways the ODBC database can receive the data: by overwriting old values with new values in a single row, or by adding a new row for each new value. These are determined by the last argument in the `.WatchPoint` function.:

  ```
  .WatchPoint (#$DataSim:Square, tableclass, nil);
  .WatchPoint (#$DataSim:Sine, tableclass);
  ```

  The default is to overwrite values. This is what happens for values pertaining to `DataSim:Sine`. To have the DataHub write a new line for each change, you can add a final argument, `nil`, such as in `DataSim:Square` above.

- **Add more DataSim points.** To add other points from DataSim, use this format for new rows:

  ```
  .WatchPoint (#$DataSim:pointname, tableclass, nil);
  ```

or this format for overwriting rows:

```
.WatchPoint (#$DataSim:pointname, tableclass);
```

Where *pointname* is the name of a point in DataSim.

- **Add your own points** You can add your own points using this syntax:

```
.WatchPoint (#$domain:pointname, tableclass, nil);
.WatchPoint (#$domain:pointname, tableclass);
```

where *domain* is the domain that the point is in, and *pointname* is the name of the point.

2.4. Tutorial 4: Writing data from a database to the DataHub

This tutorial demonstrates how to keep the DataHub updated every second with the latest values in a database.

This tutorial uses the same DSN and database as Tutorial 2, but creates a different table called "control" (see below). If you haven't done Tutorial 2 yet, please review Getting Started in that section to see how to set up your system for this tutorial.

As with Tutorial 2, you will need to create a table in the database. This new table should be named "control", and should contain at least three columns with names, data types, and other attributes exactly as specified here:

Column name	Data type	Other attributes
ID	integer	identity, non-null, counter
CTRLNAME	text string	null
CTRLVALUE	real	non-null

Any other columns in this table must be allowed to take on a null value.

Once this script is running, you can enter the name of any existing DataHub point in a row of the database in the CTRLNAME column. Make sure you enter the full point name, including the domain name, with the syntax *domainname*:*pointname*. Enter a corresponding value for the point in CTRLVALUE. The entered value will appear for that point in the DataHub. Any time the value changes in the database, the results get passed to the DataHub within a second. The point in the DataHub will continue to be updated once every second from the database as long as the two are both running.

The complete code for this tutorial is shown below, and is included in your DataHub distribution, in the scripts subdirectory, such as C:\Program Files\Cogent\DataHub\scripts*myscript.g*. Please refer to Accessing Scripts in the DataHub Scripting manual for details on how to load and run a script.

The Code: ODBCTutorial4.g

```
/* All user scripts should derive from the base "Application" class */

require ("Application");

/* Get the Gamma library functions and methods for ODBC and/or
 * Windows programming.  Uncomment either or both. */

require ("ODBCSupport");
```

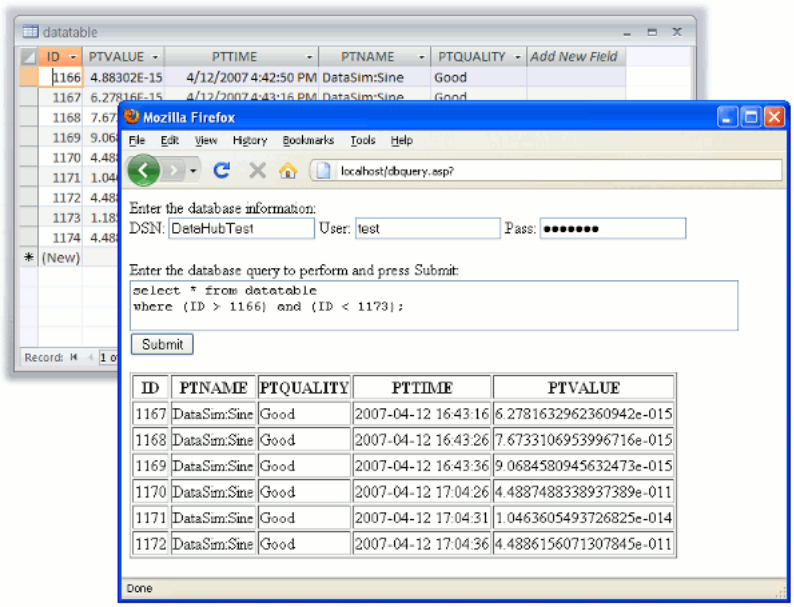
```
/* Applications share the execution thread and the global name
 * space, so we create a class that contains all of the functions
 * and variables for the application.  This does two things:
 *   1) creates a private name space for the application, and
 *   2) allows you to re-load the application to create either
 *      a new unique instance or multiple instances without
 *      damaging an existing running instance.
 */

/*
 * This application assumes that the table specified by the "tablename"
 * member variable exists in the DSN specified by the "DSN" member
 * variable below.
 * The table consists of at least the following columns:
 *    ID - integer, identity, non-null, counter
 *    CTRLNAME - text string, null
 *    CTRLVALUE - real, non-null
 * Any other columns in this table must be allowed to take on a
 * NULL value.
 */

class ODBCTutorial4 Application
{
    /* User-defined values, may be changed as needed. */
    DSN = "DataHubTest";
    user = "test";
    password = "test";
    tablename = "Table1";

    /* These values get defined by the program.*/
    conn;
    env;
    tableclass;
    is_connected;
    is_connecting;
}

/* Connect to the DSN and create a class that maps the table. */
method ODBCTutorial4.Connect ()
{
    local   ret;
    .is_connecting = t;

    protect
    {
        /* Create the ODBC environment and connection */
        if (!.env)
            .env = ODBC_AllocEnvironment();
        if (!.conn)
            .conn = .env.AllocConnection();

        /* Attempt the connection. */
        ret = .conn.Connect (.DSN, .user, .password);
        if (ret != SQL_SUCCESS && ret != SQL_SUCCESS_WITH_INFO)
            error (.conn.GetDiagRec());

        /* Create a class from the table */
        .tableclass = .conn.ClassFromTable (#DataEntry, nil, .tablename);

        /* Set the primary key.  This is redundant for MS-SQL and MYSQL since
           they can figure it out themselves, but Access requires it. */
        .conn.SetPrimaryKey (.tableclass, "ID");

     .is_connected = t;
    }
    unwind
    {
```

```
        .is_connecting = nil;
        if (.is_connected)
            princ ("    Connection successful\n");
        else
            princ ("    Connection failed\n");
    }
}

/* Disconnect from the database server */
method ODBCTutorial4.Disconnect ()
{
    if (.conn)
    {
        try
        {
            princ ("Disconnecting database\n");
            .conn.Disconnect();
            destroy(.conn);
        }
        catch
        {
            princ ("Disconnection failed: ", _last_error_, "\n");
        }
        .conn = nil;
    }
    .is_connected = nil;
}

/* Try to reconnect to the database if it's not currently connected. */
method ODBCTutorial4.Reconnect()
{
    if (!.is_connected && !.is_connecting)
    {
        .Connect();
    }
}

/* Reload information from a database record into the DataHub.  This
   is being called on a timer.  We re-query the database for the
   given record, then update the DataHub points simply by assigning
   them. */
method ODBCTutorial4.Update ()
{
    local  result;

    if (.is_connected)
    {
        try
        {
            result = .conn.QueryToClass (.tableclass, string ("select * from ", .tablename));
            with x in result do
            {
               datahub_write (x.CTRLNAME, x.CTRLVALUE);
            }
        }
        catch
        {
            /* If the query fails then disconnect and do not try again until a
               successful connection has been made based on the reconnect timer. */
            princ("Query failed.  Disconnecting - ", _last_error_, "\n");
            .Disconnect();
        }
    }
}

/* The mainline.  Connect to the database and begin storing data from
   the DataHub into the database. */
method ODBCTutorial4.constructor ()
```

```
{
    /* Every second, read the 'control' database, and update the values.
       This keeps the value in the DataHub always in sync with the database.
       The timer value can be fractional, such as 0.5 for twice per second.  */
    .TimerEvery (1, '(@self).Update ());

    /* Start a timer that will reconnect to the database every 5 seconds
       if the connection has been lost. */
    .TimerEvery(5, '(@self).Reconnect());

    /* Try connecting now.  If this fails, the timer will try again later. */
    .Reconnect();

}

/* Any code to be run when the program gets shut down. */
method ODBCTutorial4.destructor ()
{
    .Disconnect();
    if (.env)
        destroy(.env);
}

/* Start the program by instantiating the class.  If your
 * constructor code does not create a persistent reference to
 * the instance (self), then it will be destroyed by the
 * garbage collector soon after creation.  If you do not want
 * this to happen, assign the instance to a global variable, or
 * create a static data member in your class to which you assign
 * 'self' during the construction process.  ApplicationSingleton()
 * does this for you automatically. */
ApplicationSingleton (ODBCTutorial4);
```

2.5. Viewing data from a web browser

Using the DataHub Web Server, it is possible to send an SQL query from a web page to a database, and view the results in the web page.



Please refer to the Web Server documentation in the Cogent DataHub manual for more information.

Chapter 3. An Explanation of the Tutorial Code

The DataHub interacts with relational databases through ODBC (Open Data Base Connectivity). Every database acts a little differently, and each person's requirements are specific to the their own application. Consequently, the DataHub offers its ODBC support through scripts, written in the Gamma language.

A DataHub script consists of four parts:

- Define the application object
- Interactions with the database
- Set up event handlers
- Shut down

3.1. Define the Application Object

A DataHub script defines an object called an `Application` object. This object is a class derived from the class `Application`. All of the functions that you define should be methods of this class. Variables should be members of this class wherever possible.

```
class MyODBCScript Application
{
  env;    // store the ODBC environment here
  conn;   // store the ODBC connection here
}
```

This defines the application object.

The application object requires a constructor. The constructor acts as the main line of your program. To start your program, you just create an instance of the application class, causing the constructor to run.

```
method MyODBCScript.constructor ()
{
  // This is the program main line
}
```

The job of the constructor is to define event handlers. An event handler is program code attached to a particular event. An event can be one of:

- A change in a DataHub point value
- A timer
- A Microsoft Windows event

The constructor runs to completion. Once the constructor completes, the script engine begins processing events, and executes the program code attached to those events.

The script engine will continue to process events until the application object is destroyed. You may optionally provide a destructor for the object to clean up any specific data or open any files or timers that your application has created.

```
method MyODBCScript.destructor ()
{
  // This is where you clean up open files and connections
}
```

You can add other methods to the application object as you need them. Only the constructor is necessary.

3.2. Interactions with the Database

3.2.1. Connecting to the Database

Connecting to an ODBC database is a three-step process:

1. Create an `ODBCEnvironment` object.
2. Create an `ODBCConnection` object.
3. Connect to a DSN (Data Source Name).

```
.env = ODBC_AllocEnvironment();
.conn = .env.AllocConnection();

/* Attempt to connect. */
ret = .conn.Connect ("DSN name", "user name", "password");
if (ret != SQL_SUCCESS && ret != SQL_SUCCESS_WITH_INFO)
    error (.conn.GetDiagRec());
```

If the connection fails, the return value will be something other than `SQL_SUCCESS` or `SQL_SUCCESS_WITH_INFO`. You can query the database for details of the failure by calling the `GetDiagRec` method of the `ODBCConnection`.

3.2.2. Creating a Gamma Class from a Database Table

The tutorial code connects to an existing database table. If you would like the script create a table for you, please refer to Section 3.2.6, *Creating a Database Table* below.

ODBC tables are mapped as classes into Gamma. This means that most interactions with the database will be through convenient method calls rather than SQL queries. Gamma is able to automatically determine the structure of a table in your database and to create a class from it:

```
tableclass = .conn.ClassFromTable (#ClassName, nil, "table_name");
```

This statement will look up the table named `table_name` in your database, and create a class whose name is `ClassName` from it. It will also return the class definition into the variable `tableclass`.

Gamma attempts to determine the primary key field from your table. Some databases, such as Microsoft Access, do not provide a facility to do this. In that case, you need to assign the primary key for the table:

```
.conn.SetPrimaryKey (tableclass, "id");
```

The resulting class will have a number of special data members, like `__table`, the name of the table, as well as a data member for each column of the database table.

3.2.3. Querying Rows from the Database

Once you have mapped your database table to a Gamma class, you can query the database by constructing a `select` SQL call:

```
allrows = .conn.QueryToClass (tableclass, string ("select * from ", tableclass.__table));
```

This statement will query all of the rows in the table attached to `tableclass`, and return them as an array in `allrows`. Each element of the array will be a class instance.

3.2.4. Inserting Rows into a Database

To insert a row, first create a member of the class associated with the table, and then use `ODBCConnection.Insert` to insert it:

```
local   timestamp;
local   record;

record = new (tableclass);
timestamp = localtime (clock());
timestamp = format ("%d-%02d-%02d %02d:%02d:%02d",
        timestamp.year + 1900, timestamp.mon + 1,
        timestamp.mday, timestamp.hour, timestamp.min,
        timestamp.sec);
record.ptname = "point name";
record.ptvalue = point_value;
record.pttimestamp = timestamp;
record.ptquality = point_quality;
.conn.Insert (record);
```

This code creates a record to be inserted, assigns a value to each column in the record, and then inserts it into the database. Gamma will attempt to convert the values in each column to the type required by the database. If an error occurs, the `Insert` method will throw an error that can be handled by a `try/catch` block.

You can also insert rows by constructing your own SQL statement and submitting it using the `ODBCConnection.QueryAndStore` method.

3.2.5. Updating Existing Rows in a Database

To update an existing row in a database, you must know the primary key for the row you wish to update. Normally you find this key by performing a query on the database:

```
local   result = .conn.QueryToClass (tableclass, string
                                     ("select * from ",
                                      tableclass.__table,
                                      " where name = '",
                                      "the_point_name", "'"));
local   record = result[0];
```

The primary key column of the record will identify the row in the database. You can now modify the record:

```
record.ptvalue = new_point_value;
```

and update the record in the database:

```
.conn.Update (record);
```

If an error occurs, the `Update` method will throw an error that can be handled by a `try/catch` block.

3.2.6. Creating a Database Table

In most cases, you will create the database table using your database management software. This gives you a more convenient interface to the creation process. Different databases use different syntax to create a table. However, if you need to create a table within your script, there are two options:

1. Call the `ODBCConnection.CreateTable` method.
2. Call the `ODBCConnection.QueryAndStore` method.

The `CreateTable` method helps to construct the query, but it still requires you to understand the SQL syntax of table creation for your database.

```
.conn.CreateTable ("table name",
                   "id int PRIMARY KEY IDENTITY",
                   "ptname VARCHAR(20) NOT NULL",
                   "ptvalue DOUBLE NOT NULL",
                   "pttimestamp DATETIME NOT NULL",
```

```
                    "ptquality VARCHAR(20) NOT NULL"
                    );
```

The `CreateTable` arguments consist of the table name followed by any number of definitions for the columns in the table. The column definitions depend on the database being used. In particular, the primary key field (in this example, `id`) is very different from one database to another. The primary key must be integer, unique and auto-incrementing. In the case of Microsoft Access, you must issue an additional query to create a primary key:

```
.conn.CreateTable ("table_name",
                    "id COUNTER",
                    "ptname VARCHAR(20) NOT NULL",
                    "ptvalue DOUBLE NOT NULL",
                    "pttimestamp DATETIME NOT NULL",
                    "ptquality VARCHAR(20) NOT NULL"
                    );
.conn.QueryAndStore ("create unique index p_id on table_name (id)
                      with primary disallow null");
```

The alternative to using `ODBCConnection.CreateTable` is to construct your own complete SQL statement and submit it to the database using `ODBCConnection.QueryAndStore`:

```
.conn.QueryAndStore ("create table table_name
                      (id COUNTER, ptname VARCHAR(20) NOT NULL,
                                   ptvalue DOUBLE NOT NULL,
                                   pttimestamp DATETIME NOT NULL,
                                   ptquality VARCHAR(20) NOT NULL)");
.conn.QueryAndStore ("create unique index p_id on table_name (id)
                      with primary disallow null");
```

If any call to `CreateTable` or `QueryAndStore` generates an error, then the error will be thrown. You can catch the error using a `try/catch` block within your script.

3.3. Set up Event Handlers

The Gamma application will normally define functions to be called when an event occurs. In ODBC applications, this is most commonly to write live data into the relational database. Another word for an event is a "trigger". It is up to you to decide the best trigger for writing data into the database.

An event handler will look something like this:

```
.OnChange (#$DataSim:Sine, '(@self).AddRecord (this, value));
```

This statement says that when the DataHub point `DataSim:Sine` changes, we want to call our application's `AddRecord` method with two arguments. The special variables `this` and `value` are always defined to be the point name and the point value respectively when running an `OnChange` handler. We might define our `AddRecord` method like this:

```
method MyODBCScript.AddRecord (pointname, newvalue)
{
    local  record = new (tableclass);
    local  timestamp;

    timestamp = localtime (PointGetUnixTime(pointname));
    timestamp = format ("%d-%02d-%02d %02d:%02d:%02d",
        timestamp.year + 1900, timestamp.mon + 1,
        timestamp.mday, timestamp.hour, timestamp.min,
        timestamp.sec);
    record.ptname = string (pointname);
    record.ptvalue = number (newvalue);
    record.pttimestamp = timestamp;
    record.ptquality = GetQualityName(PointMetadata(pointname).ptquality);
    .conn.Insert (record);
    record;
}
```

If you would like to add a record at a specific interval rather than on a value change, you could create a timer:

```
timerid = every (0.5, '(@self).AddRecord (#$DataSim:Sine, $DataSim:Sine));
```

This statement will add a new record to the database every ½ second. In this case, we need to specify the name of the point, `$DataSim:Sine`, as part of the `AddRecord` call because the special variables `this` and `value` are not defined during a timer call. The first reference to `$DataSim:Sine` is protected from evaluation during the call using the # modifier, so its name is passed to the first argument when the expression is evaluated (when the timer occurs). The second reference is not protected from evaluation, so its value is passed to the second argument.

You need to keep track of outstanding timers so that you can clean them up during the destructor:

```
method MyODBCScript.destructor ()
{
  cancel (timerid);
}
```

3.4. Shut Down

Normally an `Application` is not started and stopped repeatedly. The application is normally started when the DataHub starts, and continues until the DataHub is stopped.

To shut down your application, you need to destroy the application instance. This will cause the destructor to be called. Your clean-up code should be located in the destructor. The `Application` class automatically cleans up event handlers created using `OnChange`. You can detach from the ODBC database in your destructor:

```
method MyODBCScript.destructor ()
{
    .conn.Disconnect();
    cancel (timerid);
}
```

If you do not disconnect from the database in the destructor, Gamma will disconnect from the database when its garbage collector destroys the `ODBCConnection` object.

You can destroy the application instance by calling the `destroy` function. This can be triggered by a data point change or some other method. For example:

```
.OnChange (#$default:kill_application, 'destroy(@self));
```

This statement would wait for a change to the DataHub point named `default:kill_application` and then destroy the application. The application could then be restarted from the Scripting option in the DataHub Properties window.

Chapter 4. Multi-Threaded ODBC Interface

The DataHub includes a scripting interface to multi-threaded database access. This comes in addition to the existing single-threaded ODBC implementation. A multi-threaded interface is substantially different from a single-threaded interface, and offers a number of advantages:

1. **Non-blocking:** A single-threaded interface can block when attempting to communicate with a database that is no longer available. While blocked, all other Gamma scripts are halted until the database responds or the connection times out. This has a particularly substantial impact when attempting to communicate with more than one database at a time, since blocking on one database will also halt communication to the other database.

   With a multi-threaded interface, a separate thread is spawned to communicate with each database. If a communication problem occurs, only the database thread blocks. All other Gamma scripts continue to run normally.

2. **Store-and-forward:** A single-threaded interface must be willing to drop data when the database becomes temporarily too busy to handle incoming data, or when the database is disconnected for some reason, such as a network failure.

   A multi-threaded interface can store data to disk when the database is unresponsive or unavailable. When the database recovers, the stored data can be transmitted to the database. Some multi-threaded implementations do not preserve operation order. Gamma's implementation guarantees that every operation will be performed in the same order that it was originally attempted.

3. **Time-delayed write:** The multi-threaded implementation can store data to disk, to be written to the database at a later time or in batches as defined intervals. This is particularly useful if bandwidth is limited at certain times of day, or where the database connection is expensive to maintain at all times. For example, a database connection using satellite Internet could be charged based on connection time. It is much more efficient to store data temporarily and then connect at a predefined interval to update the database.

4.1. How-To

To ensure that the primary script thread does not block when there is a communication problem with the the database, all communication to the database must be asynchronous. This means that a database command will normally not produce an immediate result. The result will be delivered at some time in future by triggering a "callback" function in the script. This is a different way of thinking about a script, and takes a little getting used to. Gamma provides a number of functions to make this as simple as possible.

4.1.1. Create an ODBCThread Instance

All communication with the database is performed through an instance of the class `ODBCThread`. For example:

```
mysql = new ODBCThread();
flags = STORE_AND_FORWARD;
cachefile = "C:/temp/mysql.cache";
mysql.Configure ("myDSN", "username", "password", flags, cachefile);
```

This code is sufficient to create the database connection. At this point, the database is not connected, and the thread that handles the database has not yet been started.

4.1.2. Attach Event Callbacks

The next step is to attach the callback functions that will alert the script to various events from the database thread. Whenever a callback function is executed, a special variable called `SQLResult` will be defined for the duration of the callback. This contains information about the result of the SQL command, a description of the command and error codes. This information is stored in an instance of the class `ODBCThreadResult`.

A callback is attached by simply assigning the code to run to the callback member of the `ODBCThread` object. For example, to print information when the connection fails, you could do this:

```
method MyApp.onConnectionFail()
{
    princ ("Connection closed: ", SQLResult.Description, "\n");
}
```

and then in the application constructor, do this:

```
thread.OnConnectionFailed = '(@self).onConnectFail();
```

The following callbacks are defined:

`OnConnectionSucceeded`

Called when the connection to the database transitions from not connected to connected.

`OnConnectionFailed`

Called when the connection to the database transitions from connected to not connected, or when a connection attempt fails. The `Description` member of the `SQLResult` contains information about the reason for the event.

`OnExecuteStored`

Called when a stored transaction is successfully forwarded to the database. The `SQLResult` contains information about the transaction.

`OnFileSystemError`

Called when a file system error occurs when reading or writing to the disk. The `Description` member for the `SQLResult` has details about the error.

`OnODBCError`

Called when an ODBC error occurs that cannot be reported as part of the result set from a successful transaction. Typically this would be a failure generated by an attempt to execute a stored transaction.

4.1.3. Configure Startup Actions

One of the most difficult concepts of using an asynchronous interface is the idea that a sequence of steps must be performed in order during startup, even though the script code cannot be executed sequentially. At each step, there could be an asynchronous request/event pair that breaks up the sequence into disjoint code fragments.

The `ODBCThread` class defines a startup state machine that helps to sequence these steps. As a script developer, you define the steps that will be performed by creating a series of initialization steps, or "stages" that will be performed after a successful database connection. These stages will not automatically be executed. It is up to you to initiate them. This allows you to choose whether to execute

these stages on each connection, or only on the first connection, or to start executing the stages at a stage other than the first.

To create initialization stages, make repeated calls to the method "addInitStage":

```
AddInitStage(sqlCommand, onSuccess, onFailure);
```

The *sqlCommand* argument contains either a string containing valid SQL, or inline code to be executed. If the *sqlCommand* is a string, it is passed to the database thread for execution. If it is inline code, it will be executed immediately.

The *onSuccess* and *onFailure* arguments are inline code that will be executed when the *sqlCommand* has returned a result. Only one of these two will be executed for each execution of *sqlCommand*. The return value from *onSuccess* is ignored. If the return value from *onFailure* is `nil`, then the initialization sequence is aborted. It is sufficient to supply `nil` to *onFailure* to abort the sequence on any error, and to supply `t` to *onFailure* to continue the sequence if there is an error.

For example, the following sequence could be used:

```
// Drop the existing table.  If we get an error, continue
   thread.AddInitStage("drop table mytable", nil, t);

// Create a new table.  Once the table is created, set up an
// auto-incrementig primary key.  If either the table creation or
// the key definition fails, abort.
thread.AddInitStage("create table mytable ( myid counter, myvalue number, tstamp number )", nil, nil);
thread.AddInitStage("create unique index p_myid on mytable (myid) with primary disallow null", nil, nil);

// Make a call to getTableInfo to look up all tables in the
// database.  Since getTableInfo is asynchronous, we must use its
// callback to resume the initialization sequence by calling the
// special method "cbInitStage()" in the callback, and then
// letting the initialization sequencer call the real callback
// mapTable is an example of user-created code.
thread.AddInitStage('(@thread).getTableInfo("", "", "", "TABLE", '(@thread).cbInitStage()),
                    '(@self).mapTable(@thread, "mytable", SQLTables), nil);

// We happen to know that the database we use does not provide
// primary key information so we have to set the primary key
// manually after the table has been mapped to  class.
thread.AddInitStage(nil, '(@self).SetClassKey((@self).tableclass, #myid), nil);

// Finally, we have passed through all the initialization steps,
// so we begin storing data.  beginDataStream in this example is
// user-generated code, which might set up timers or data event
// handlers that cause the script to write data to the database.
thread.AddInitStage(nil, '(@self).beginDataStream(@thread), nil);
```

Notice that the final two stages do not define the *sqlCommand* at all. This will cause their success code to run immediately and then to move on to the next initialization stage. If an initialization stage has no *sqlCommand*, it cannot fail.

At this point, the initialization stages are defined but have not run. They will not be run until the script calls:

```
thread.beginAsyncInit();
```

Commonly, you only want to run the initialization once, so you might handle it in the `OnConnectionSuccess` handler like this:

```
method MyApp.onConnect(thread)
{
    princ ("Connection succeeded\n");
    if (thread.is_first_connect)
    {
        thread.beginAsyncInit();
    }
}
```

4.1.4. Start the Database Thread

Once you have defined the parameters for the database connection, defined the callback handlers and defined the initialization code, the connection is completely defined, but still is not running. To start the thread and begin connecting, call the `Start` method:

```
thread.Start();
```

At this point the thread is started and begins trying to connect to the database. As the thread runs it may call your callback functions in any order to indicate successes, failures, and errors.

4.2. Store and Forward

Store and Forward is a term used to describe a database connection where the data is stored locally to disk and then later forwarded to the database. The `ODBCThread` object performs an advanced form of store and forward that does more than simply store data for later delivery.

For any SQL statement given to the `ODBCThread.ExecDirect` method, you can optionally specify that this particular statement should be stored for later forwarding. Normally these will be `INSERT` or `UPDATE` statements, but they could be an SQL statement that must be executed at the database, such as stored procedures or `ALTER` statements. The `ODBCThread` guarantees that all storable SQL commands will be executed in the order in which they are specified, even if they are first stored to file.

The `ODBCThread` object uses a sophisticated store and forward technique that only writes to disk if the database is not connected, or has been paused. If the database is available, the commands will be transmitted directly to the database. This means that there is no penalty to using store and forward during normal operation.

`ODBCThread` also maintains an in-memory queue of pending operations. This queue helps to avoid writing to disk during busy periods or during short database or network outages. You can modify the depth of this queue to reduce the chance of involving the disk during busy periods. The queue depth defaults to 100 messages, but it can be modified by setting the `MaxTransactions` member of the ODBCThread. For example:

```
thread.MaxTransactions = 200;
```

For the thread to perform store and forward, the flag `STORE_AND_FORWARD` must be specified when initially configuring the thread, and the flag `STORE_AND_FORWARD` must also be specified for any call to `ExecDirect` that should be a candidate for store and forward treatment.

4.2.1. Time Delayed Writes

Time delayed writing is used to avoid maintaining a continuous connection to the database, or to make use of times of day where the network utilization is low. You can call the `Pause` and `Disconnect` methods at any time to cause the thread to pause output to the database, then disconnect. All further transactions will be written to the local disk cache until the database is reconnected. To resume writing to the database, the script just needs to call the `Resume` method. The database thread will automatically reconnect to the database when it can.

You can periodically test to see whether the disk cache has been transmitted by running a repeating timer that calls `CacheIsEmpty`. When `CacheIsEmpty` returns non-nil, the disk cache has been consumed. At this point the script can once again `Pause` and `Disconnect` the thread.

Using this method, the script can transmit the cached data based on the time of day, a process event, or even input from an operator.

4.3. Example

```
/*
 * This script demonstrates the use of the threaded ODBC interface to insert
 * data into a database based on a timer.
 */

require ("Application");
require ("ODBCThreadSupport");
require ("Time");
require ("Quality");

class ODBCThreadDemo Application
{
    DSN = "MySQLLocal";        // The DSN name to use for the database connection
    username = "root";          // The user name for connecting to the database
    password = "GY&*ik";          // The password for connecting to the database
    tablename = "andrew5";      // The name of the database table
    cachefile = "c:/tmp/testcache.txt";     // Base name for the disk cache file

    tableclass;
    thread;
}

/* This method will be called every time the connection is established to the database.
 * If there is something we only want to perform on the first connection, we can test
 * is_first_connect to perform the code only once.
 */
method ODBCThreadDemo.onConnect()
{
    princ ("Connection succeeded\n");
    if (.thread.is_first_connect)
    {
        // Start the sequence defined by the AddInitStage calls in the constructor
        .thread.BeginAsyncInit();
    }
}

/* If we get a connection attempt failure, or the connection fails after having been
 * connected, this method is called.
 */
method ODBCThreadDemo.onConnectFail()
{
    princ ("Connection closed: ", SQLResult.Description, "\n");
}

/* Map the table in the set of table definitions that matches the name in .tablename
 * into a Gamma class.  This lets us easily convert between class instances and rows
 * in the table.
 */
method ODBCThreadDemo.mapTable(name, tabledefinitions)
{
    .tableclass = .thread.ClassFromTable(name, tabledefinitions);
}

/* Set up the timer or event handler functions to write to the table. */
method ODBCThreadDemo.startLogging()
{
 /* You can modify and/or add similar timers or event handlers for
  * for each data point that you want to log.  Please refer to the
  * "Methods and Functions from Application.g" section of the documentaton
  * for more details about the timer and event handler funtions.
  * http://www.cogentdatahub.com/Docs/dhs-reference-applicationg.html
  */
 // Log a new row of data every 3 seconds.
    .TimerEvery(3, '(@self).writeData(#$DataSim:Sine));

 // Log a new row of data at 20 seconds past each minute of each hour, etc.
```

```
    .TimerAt(nil, nil, nil, nil, nil, 20, '(@self).writeData(#$DataSim:Triangle));

    // Log a new row of data for the point DataSim:Square when it changes.
    .OnChange(#$DataSim:Square, '(@self).writeData(this));

    // Log a new row of data for the point DataSim:Sine when DataSim:Square changes.
    .OnChange(#$DataSim:Square, '(@self).writeData(#$DataSim:Sine));
}

method ODBCThreadDemo.writeData(pointsymbol)
{
    local       row = new (.tableclass);
    local       pttime, ptltime;
    local       timestring;

    // Generate a timestamp in database-independent format to the millisecond.
    // Many databases strip the milliseconds from a timestamp, but it is harmless
    // to provide them in case the database can store them.
    pttime = WindowsTimeToUnixTime(PointMetadata(pointsymbol).timestamp);
    ptltime = localtime(pttime);
    timestring = format("{ts '%04d-%02d-%02d %02d:%02d:%02d.%03d'}",
        ptltime.year+1900, ptltime.mon+1, ptltime.mday, ptltime.hour, ptltime.min, ptltime.sec,
        (pttime % 1) * 1000);

    // Fill the row.  Since we mapped the table into a Gamma class, we can access
    // the columns in the row as member variables of the mapped class.
    row.ptname = string(pointsymbol);
    row.ptvalue = eval(pointsymbol);
    row.pttime = timestring;
    // Perform the insertion.  In this case we are providing no callback on completion.
    .thread.Insert(row, nil);
}

/* Write the 'main line' of the program here. */
method ODBCThreadDemo.constructor ()
{
    // Create and configure the database connection object
    .thread = new ODBCThread();
    .thread.Configure(.DSN, .username, .password, STORE_AND_FORWARD, .cachefile, 0);

    // Use this to delete the table on the first connection after the script starts.
    // BE CAREFUL - re-running the script will start over and delete the table again.
    // .thread.AddInitStage(format("drop table %s", .tablename), nil, t);

    // Use this to create the table if it does not exist.  Note: this might not work for all databases.
    // When in doubt, create the table manually.  The 't' in the onFail argument says to ignore errors
    // and continue with the next stage.
    // .thread.AddInitStage(format("create table %s (ptid int auto_increment primary key, ptname varchar(64),
    //                             ptvalue double, pttime datetime )", .tablename), nil, t);

    // Query the table and map it to a class for eash insertion.  We want to run an asynchronous event
    // within the asynchronous initialization stage, so to do that we specify the special method
    // cbInitStage as the callback function of our asynchronous event (GetTableInfo).  We deal with
    // the return from the GetTableInfo in the onSuccess argument of the init stage.
    .thread.AddInitStage('(@.thread).GetTableInfo("", "", (@.tablename), "TABLE,VIEW",
                                                '(@.thread).cbInitStage()),
                         '(@self).mapTable(@.tablename, SQLTables), nil);

    // Do not start writing data to the table until we have successfully created and mapped
    // the table to a class.  If we wanted to start writing data immediately, then we would
    // create the table class beforehand instead of querying the database for the table
    // definition.  Then, even if the database were unavailable we could still cache to the
    // local disk until the database was ready.
    .thread.AddInitStage(nil, '(@self).startLogging(), nil);

    // Set up the callback functions for various events from the database thread
    .thread.OnConnectionSucceeded = '(@self).onConnect();
    .thread.OnConnectionFailed = '(@self).onConnectFail();
```

```
    .thread.OnFileSystemError = ‘princ("File System Error: ", SQLResult, "\n");
    .thread.OnODBCError = ‘princ("ODBC Error: ", SQLResult, "\n");
    .thread.OnExecuteStored = nil;

    // Now that everything is configured, start the thread and begin connecting.  All of the
    // logic now will be driven through the onConnect callback and then through the init
    // stages.
    .thread.Start();

}

/* Any code to be run when the program gets shut down. */
method ODBCThreadDemo.destructor ()
{
}

/* Start the program by instantiating the class. */
ApplicationSingleton (ODBCThreadDemo);
```

Chapter 5. Classes

DATE_STRUCT

DATE_STRUCT — contains dates (y,m,d).

Synopsis

```
class DATE_STRUCT
{
    day;
    month;
    year;
}
```

Description

This structure contains dates. For more information, please refer to C Data Types.

ODBCColumn

ODBCColumn —

Synopsis

```
class ODBCColumn
{
    columnsize;
    datatype;
    decimaldigits;
    name;
    nullable;
}
```

Description

Not yet documented.

ODBCConnection

ODBCConnection — allocates a connection handle.

Synopsis

```
class ODBCConnection ODBCHandle
{
    __stmt;
    h;
    handle;
    type;
}
```

Base Classes

ODBCHandle <-- ODBCConnection

Description

This class allocates a connection handle. It corresponds to using the value SQL_HANDLE_DBC for the *HandleType* of the SQLAllocHandle function.

Class Members

These functions are identical to the corresponding C or C++ functions, as noted.

`AddColumn (tablename, column, type, default_val, allow_null)`

corresponds to AddColumn.

`AllocDescriptor ()`

corresponds to AllocDescriptor.

`AllocStatement ()`

corresponds to SQLAllocStmt.

`ClassFromColumns (symclassname, superclass, columns)`

corresponds to ClassFromColumns.

`ClassFromTable (symclassname, superclass, tablename)`

corresponds to ClassFromTable.

`ClassesFromTables (superclass, verbose?=nil)`

corresponds to ClassesFromTables.

`Connect (ServerName, UserName, Authentication)`

corresponds to SQLConnect.

`CreateTable (tablename, columns...)`

corresponds to CreateTable.

`Delete (row)`

corresponds to Delete.

`Disconnect ()`

corresponds to SQLDisconnect.

`DropTable (tablename)`

corresponds to DropTable.

`EndTran (CompletionType)`

corresponds to SQLEndTran.

`Error ()`

corresponds to SQLError.

`GetOneColumn (query_string)`

corresponds to GetOneColumn.

`GetOneValue (query_string)`

corresponds to GetOneValue.

`Insert (row)`

corresponds to Insert.

`MakeClass (symclassname, superclass, ivars, tablename, primary_key)`

corresponds to MakeClass.

`MapClassFromResponse (klass, response)`

corresponds to MapClassFromResponse.

`Query (query_string)`

corresponds to Query.

`QueryAndStore (query_string)`

corresponds to QueryAndStore.

`QueryToClass (klass, query_string)`

corresponds to QueryToClass.

`QueryToTempClass (superclass, query)`

corresponds to QueryToTempClass.

`ReQuery (row)`

corresponds to ReQuery.

`Statement ()`

corresponds to Statement.

`StoreResult ()`

corresponds to StoreResult.

`Update (row)`

corresponds to Update.

(The following functions are inherited from: ODBCHandle)

`GetDiagRec ()`

corresponds to SQLGetDiagRec.

ODBCDescriptor

ODBCDescriptor — allocates a descriptor handle.

Synopsis

```
class ODBCDescriptor ODBCHandle
{
    h;
    handle;
    type;
}
```

Base Classes

ODBCHandle <-- ODBCDescriptor

Description

This class allocates a descriptor handle. It corresponds to using the value SQL_HANDLE_DESC for the *HandleType* of the SQLAllocHandle function.

ODBCEnvironment

`ODBCEnvironment` — allocates an environment handle.

Synopsis

```
class ODBCEnvironment ODBCHandle
{
    h;
    handle;
    type;
}
```

Base Classes

`ODBCHandle` <-- `ODBCEnvironment`

Description

This class allocates an environment handle. It corresponds to using the value `SQL_HANDLE_ENV` for the *`HandleType`* of the `SQLAllocHandle` function.

Class Members

These functions are identical to the corresponding C or C++ functions, as noted.

`AllocConnection ()`

corresponds to SQLAllocConnect.

(The following functions are inherited from: ODBCHandle)

`GetDiagRec ()`

corresponds to SQLGetDiagRec.

ODBCHandle

ODBCHandle — a parent class for connections, descriptors, environments, and statements.

Synopsis

```
class ODBCHandle
{
    handle;
    type;
}
```

Description

This class is a parent class for ODBCConnection, ODBCDescriptor, ODBCEnvironment, and ODBCStatement handles. Together, these classes provide the functionality of SQLAllocHandle.

Class Members

These functions are identical to the corresponding C or C++ functions, as noted.

GetDiagRec ()

corresponds to SQLGetDiagRec.

ODBCResult

ODBCResult —

Synopsis

```
class ODBCResult
{
    columns;
    rows;
}
```

Description

Not yet documented.

Class Members

These functions are identical to the corresponding C or C++ functions, as noted.

`ColumnIndex (column_name)`

corresponds to ColumnIndex.

ODBCStatement

ODBCStatement — allocates a statement handle.

Synopsis

```
class ODBCStatement ODBCHandle
{
    h;
    handle;
    type;
}
```

Base Classes

ODBCHandle <-- ODBCStatement

Description

This class allocates a statement handle. It corresponds to using the value SQL_HANDLE_STMT for the *HandleType* of the SQLAllocHandle function.

Class Members

These functions are identical to the corresponding C or C++ functions, as noted.

`Cancel ()`

corresponds to SQLCancel.

`CloseCursor ()`

corresponds to SQLCloseCursor.

`Columns (CatalogName, SchemaName, TableName, ColumnName)`

corresponds to SQLColumns.

`ExecDirect (StatementText)`

corresponds to SQLExecDirect.

`Execute ()`

corresponds to SQLExecute.

`Fetch ()`

corresponds to SQLFetch.

`FetchScroll (FetchOrientation, FetchOffset)`

corresponds to SQLFetchScroll.

`FreeStmt (Option)`

corresponds to SQLFreeStmt.

`GetResultData ()`

corresponds to SQLGetData.

`Prepare (StatementText)`

corresponds to SQLPrepare.

`PrimaryKeys (CatalogName, SchemaName, TableName)`

corresponds to SQLPrimaryKeys.

`RowCount (StatementText)`

corresponds to SQLRowCount.

`Tables (CatalogName, SchemaName, TableName, TableType)`

corresponds to SQLTables.

(The following functions are inherited from: ODBCHandle)

`GetDiagRec ()`

corresponds to SQLGetDiagRec.

ODBCThread

`ODBCThread` — configures the multi-threaded ODBC interface.

Synopsis

```
class ODBCThread
{
    Commands;              // Internal variable
    Callbacks;             // Internal variable
    OnExecuteStored;       // callback
    OnConnectionSucceeded; // callback
    OnConnectionFailed;    // callback
    OnFileSystemError;     // callback
    OnODBCError;           // callback
    NCached;               // Internal variable
    NUncached;             // Internal variable
    NStoredPrimary;        // Number of transactions written to level 1 cache
    NStoredSecondary;      // Number of transactions written to level 2 cache
    NForwarded;            // Number of transactions read from cache
    NStoreFailPrimary;     // Number of failures while writing to level 1 cache
    NStoreFailSecondary;   // Number of failures while writing to level 1 cache
    NForwardFail;          // Number of failures while writing to the database from cache
    NCommands;             // Number of commands ever queued to thread
    NResults;              // Number of commands results ever returned from thread
    NRejected;             // Number of commands that were rejected without queueing
    ReconnectSecs;         // Number of seconds to wait between database reconnection attempts
    ForwardDelay;          // Delay in milliseconds between transactions written from cache to database
    MaxTransactions;       // Maximum number of transactions to hold on queue
}
```

Description

As of this writing, the ForwardDelay member is not implemented.

The script developer should only modify the following members:

`OnExecuteStored`
`OnConnectionSucceeded`
`OnConnectionFailed`
`OnFileSystemError`
`OnODBCError`
`ReconnectSecs`
`ForwardDelay`
`MaxTransactions`

Methods

`ODBCThread.CacheIsEmpty()`

returns non-nil if both the level 1 and level 2 cache files are empty.

`ODBCThread.Columns(`*`catalog`*`, `*`schema`*`, `*`tablename`*`, `*`columnname`*`, `*`callback`*`)`

queries the database table for its column definitions. The *`catalog`*, *`schema`*, *`tablename`* and *`columnname`* are all strings. Some of these may accept wildcard patterns depending on the database being used. When the call completes, the callback code will be executed. The result is available in the `SQLResult` for the duration of the callback.

`ODBCThread.Configure(`*`DSN`*`, `*`username`*`, `*`password`*`, `*`flags`*`, `*`storagefile`*`, `*`maxfilesize`*`)`

sets the initial configuration for the database connection. The *`flags`* parameter can be either `0` or `STORE_AND_FORWARD`. If `STORE_AND_FORWARD` is not specified then no command on this

connection will be stored, even if the individual command specifies the STORE_AND_FORWARD flag. The *storagefile* parameter specifies the name of a file to store the level 1 cache information for store and forward operation. The level 2 cache file name will be created from this file name.

The *maxfilesize* specifies the maximum number of bytes that a cache file can grow to. There can in fact be 3 cache files, each of this size, at any one time. The *maxfilesize* may be exceeded by the length of a single transaction for any file. If you set *maxfilesize* to 0, then 2.1 GB will be used. If the size exceeds this amount then it will be truncated to 2.1 GB. You may wish to intentionally limit the file size to a lower number. In a system where the data rate is always too high for the database to handle, a smaller cache file size will cause the ODBCThread to go through periods where its data is discarded while the cache file is caught up. A smaller file will make this discard/catch-up cycle faster.

Flags can be any combination of:

- STORE_AND_FORWARD - If this flag is not set, then no file storage will take place. Any transactions that cannot be written to the thread immediately will be rejected. Any transactions in the queue that cannot be delivered to the database will be discarded.
- NO_STORE_TO_SECONDARY - This tells the ODBCThread to only use the level 1 cache. If the queue between the script and the database thread becomes full, further transactions will be rejected until there is space in the queue. However, any transactions in the queue that cannot be delivered to the database will be stored in level 1 cache.
- STORE_ALWAYS - This flag tells the ODBCThread to always store a transaction do disk before sending it to the database. This will normally cause the thread to read its queue faster, and write the transactions to disk more frequently. In case of a system crash, those transactions are more likely to be recoverable when the script re-starts. This option imposes a speed penalty if the disk is slow. On systems with fast disks, this penalty is normally minimal.
- ALLOW_CACHE_RESTART - In case of a system or application crash, the ODBCThread will resume reading any disk files at the point where it left off when the application restarts. This flag tells the ODBCThread not to track its position in the disk file, and to restart at the beginning of the file during a crash recovery. This improves speed on systems with a slow disk, but means that some transactions may be sent to the database more than once. This should only be used if disk access is slow.
- NO_FLUSH_ON_WRITE - The ODBCThread normally tries to update files as soon as possible after a write to disk. This is not efficient, but it improves the chance that more transactions will be recoverable in the case of a system or application crash. Specifying this flag will cause the ODBCThread to store data in memory longer and write to disk in larger blocks. This may improve performance for systems with a slow file system, but it increases the chance that transactions will be lost if the system crashes or shuts down.

  We recommend using either:

  ```
  STORE_AND_FORWARD
  ```

  or

  ```
  STORE_AND_FORWARD | STORE_ALWAYS
  ```

  unless the impact of disk access is unacceptably high on the system.

`ODBCThread.Connect()`

forces a connection attempt, even if the thread connection timer has not expired.

`ODBCThread.DataSources(`*`type`*`)`

lists all data sources (DSNs). The type parameter can be one of:

- `SQL_FETCH_FIRST` - retrieve all DSNs.
- `SQL_FETCH_FIRST_USER` - retrieve only user DSNs.
- `SQL_FETCH_FIRST_SYSTEM` - retrieve only system DSNs.

The return value is an array of lists of the form: `("dsn_name" . "dsn_driver")`

`ODBCThread.Disconnect()`

forces the thread to disconnect from the database. If the thread is not paused then it will attempt to reconnect to the database on the next reconnect timer cycle.

`ODBCThread.ExecDirect(`*`flags, sql, callback`*`)`

executes an SQL statement on the database. Flags can be either `0` or `STORE_AND_FORWARD`. If the command cannot be executed immediately, and `STORE_AND_FORWARD` is set, and `STORE_AND_FORWARD` is also set on the thread, then the command will be stored to file and executed later. The SQL statement is a string. When the statement is executed, the callback will be called. The result is available in the `SQLResult` for the duration of the callback.

`ODBCThread.GetFlags()`

retrieves the flags set by the `.Configure` method.

`ODBCThread.GetMessageCount()`

retrieves the number of messages currently queued to the database thread.

`ODBCThread.GetResultCount()`

retrieves the number of results currently queued from the database thread to the script thread.

`ODBCThread.Insert(`*`row, callback`*`)`

performs a database `INSERT` given an instance of a class that has been mapped to a column set in the database. The row must be an instance of a class returned from `.ClassFromResultSet`, `.ClassFromTable`, or `.ClassFromThreadResult`. When the insertion is complete, the callback is executed.

`ODBCThread.QueueIsFull()`

returns non-nil if the message queue is full.

`ODBCThread.IsPaused()`

returns non-nil if the thread is paused. See the information in `ODBCThread.Pause()`.

`ODBCThread.NoOp(`*`callback`*`)`

sends a message to the database thread, and do nothing. When the message has arrived at the database thread, the method returns and runs the callback. This is a mechanism to synchronize execution in the script with actions that are queued on the database thread.

`ODBCThread.Pause()`

pauses the thread. A paused thread will continue to store data to disk to be forwarded later, but it will not perform transactions on the database. If the database is disconnected and paused, the thread will not attempt to reconnect until the thread is resumed.

`ODBCThread.PrimaryKeys(`*`catalog, schema, tablename, callback`*`)`

queries the database for the primary keys for the given *`catalog`*, *`schema`* and *`tablename`*. The result is available in the `SQLResult` when the callback is executed.

`ODBCThread.QuoteConversion(`*`head, tail, character, replacement`*`)`

is used internally.

`ODBCThread.Resume()`

resumes a thread that has been previously paused by `.Pause()`.

`ODBCThread.SlowInsert(`*`row, callback`*`)`

is an alternate (slower) method to insert data. It acts the same as the `.Insert` method, except that it recomputes the SQL statement on each insert. The `.Insert` method computes the SQL statement ahead of time.

`ODBCThread.SlowUpdate(`*`row, callback`*`)`

is an alternate (slower) method to update data. It acts the same as the `.Update` method, except that it recomputes the SQL statement on each update. The `.Update` method computes the SQL statement ahead of time.

`ODBCThread.Start()`

starts the thread and begins attempting to connect.

`ODBCThread.Stop()`

closes the connection to the database and stops the thread.

`ODBCThread.Tables(`*`catalog, schema, tablename, tabletype, callback`*`)`

queries the database for all tables matching the *`catalog`*, *`schema`*, *`tablename`* and *`tabletype`*. It calls the *`callback`* when the transaction completes. Specifying an empty string (`""`) for any argument indicates no preference. The tabletype must be one of `"TABLE"`, `"VIEW"` or `"TABLE,VIEW"`. The result of this call is available in `ODBCResult`.

`ODBCThread.Update(`*`row, callback`*`)`

performs a database `UPDATE` given an instance of a class that has been mapped to a column set in the database. The row must be an instance of a class returned from `.ClassFromResultSet`, `.ClassFromTable` or `.ClassFromThreadResult`. When the update is complete, the callback is executed. The result of this call is available in `ODBCResult`.

`ODBCThread.ValueString(`*`value`*`)`

is used internally.

`ODBCThread.AddInitStage(`*`sqlString, onSuccess, onFailure`*`)`

adds an initialization stage to the sequential set of steps to be executed as part of the initialization after the database connections is made. See the section Configure Startup Actions above. The return value from this function is an index that can be given to `.BeginAsyncInit`.

`ODBCThread.BeginAsyncInit(`*`stage`*`?=0)`

starts executing the initialization stages in the order in which they were specified. If the stage argument is non-zero, being executing from that index in set. This index is provided as the return value from `.AddInitStage`.

`ODBCThread.cbInitStage()`

is a provided callback that can be used to trigger the next initialization stage in the initialization sequence. If the stage specifies user-defined code instead of a string for the *`sqlString`* argument of `.AddInitStage`, that code must at some point call `.cbInitStage` in order for the sequence to continue.

`ODBCThread.ClassFromResultSet(`*`columnresult, keyresult, superclass`*`?=nil, `*`symclassname`*`?=#UnboundODBCThreadTableClass)`

creates a class from the table defined in the given result sets. The *`columnresult`* is the column definition for the table, and the *`keyresult`* is the result from calling `.PrimaryKeys` on the table, or the result from querying the table through the `.Tables` method. If the *`superclass`* is non-nil, the class will be derived from *`superclass`*, otherwise it will have no parent class. If *`symclassname`* is provided, that class name will be used instead of the default *`UnboundODBCThreadTableClass`*. The class produced by this call maps each column in the *`columnresult`* to a member variable in the class. In addition, information about the primary key and the source table is held in the class. Instances of this class are suitable for use with the `.Insert` method. If the table has a primary key, then instances of this class can also be used in the `.Update` method.

`ODBCThread.ClassFromTable(`*`tablename, tables, superclass`*`?=nil, `*`symclassname`*`?=#UnboundODBCThreadTableClass)`

creates a class from the table named *`tablename`* from the table set defined in *`tables`*. The *`tables`* argument is the value of `SQLTables` from a call to the `.GetTableInfo` method. See the discussion in `.ClassFromResultSet` for more information.

`ODBCThread.ClassFromThreadResult(`*`threadresult, superclass`*`?=nil, `*`symclassname`*`?=#UnboundODBCThreadTableClass)`

creates a class from an `ODBCThreadResult` instance. This instance is usually obtained from a call to `.GetTableInfo` or `.Columns`. See the discussion in `.ClassFromResultSet` for more information.

`ODBCThread.constructor()`

is the constructor for this class. Do not override the constructor for `ODBCThread` in your own code. Instead, derive a new class from `ODBCThread` and then define a constructor for your derived class.

`ODBCThread.CreateClass(`*`symclassname, superclass, ivars, tablename, primary_key`*`)`

is the low-level call made from `.ClassFromResultSet`, `.ClassFromTable` and `.ClassFromThreadResult`. It constructs the necessary code to define a class that maps its member variables to columns in a result set.

`ODBCThread.EvalSafe(`*`code`*`)`

is used internally.

`ODBCThread.GetDataSources(`*`direction`*`?=SQL_FETCH_FIRST)`

queries the ODBC subsystem for the names of all DSNs. The type parameter can be one of:

- `SQL_FETCH_FIRST` - retrieve all DSNs.
- `SQL_FETCH_FIRST_USER` - retrieve only user DSNs.
- `SQL_FETCH_FIRST_SYSTEM` - retrieve only system DSNs.

The return value is an array of lists of the form: `("dsn_name" . "dsn_driver")`

This method is the only method that calls synchronously into the ODBC subsystem. The result is available as the return value from this function. The ODBC definition states that this call will be entirely satisfied by the driver manager, and so cannot block on the database. The database does not need to be connected, and the database thread does not have to be started for this method to succeed.

`ODBCThread.GetInsertFormat (`*`klass`*`)`

constructs the SQL statement that will be issued when the `.Insert` method is called. If you change the definition of the table by calling `.SetClassKey`, then you must also issue `.GetInsertFormat` and `.GetUpdateFormat` after `.SetClassKey` completes.

`ODBCThread.GetTableInfo (`*`catalog, schema, tablename, tabletype, callback`*`)`

produces a result that will be available in the special variable `SQLTables` for the duration of the callback. `SQLTables` is an array of arrays. Each element of the result is an array of two elements containing the table name and an instance of `ODBCThreadResult` corresponding to a call to `.Columns` for that table.

`ODBCThread.GetUpdateFormat (`*`klass`*`)`

constructs the SQL statement that will be issued when the `.Update` method is called. If you change the definition of the table by calling `.SetClassKey`, then you must also issue `.GetInsertFormat` and `.GetUpdateFormat` after `.SetClassKey` completes.

`ODBCThread.HandleColumnInfo (`*`tablename, results, callback`*`)`

is used internally.

`ODBCThread.HandleFinalInfo (`*`results, callback)`*

is used internally.

`ODBCThread.HandleTableInfo (`*`results, callback)`*

is used internally.

`ODBCThread.NextInitStage ()`

is used internally.

`ODBCThread.SetClassKey (`*`klass, keysym, ignore_if_set`*`?=t)`

sets the primary key for the class specified by *`klass`*. The class is the result of a call to `.ClassFromResultSet`, `.ClassFromTable` and `.ClassFromThreadResult`. Some databases (MS-Access in particular) do not provide information about the primary keys in a table. In order for `.Update` calls to work on this type of class, the `.SetClassKey` method must be called to tell the table which column is its primary key.

ODBCThreadResult

`ODBCThreadResult` — the results of an SQL command.

Synopsis

```
class ODBCThreadResult
{
    Result;        // The SQL result set, or nil.
    KeyResult;     // The SQL result set describing the primary
                   // keys, for those commands that produce a
                   // primary key set.
    Diagnostic;    // The complete ODBC diagnostic set, if any.
    KeyDiagnostic; // The diagnostic set for the primary key query.
    Description;   // A description of the command or result.
    ReturnCode;    // A numeric SQLRESULT for an SQL command, or an
                   // "errno" return code for a file system error.
    AffectedRows;  // The number of affected rows for an SQL command,
                   // if available.
}
```

Description

The `Diagnostic` and `KeyDiagnostic` each consist of an array, where the elements of the array are themselves arrays:

- `element[0]` = the database diagnostic message, as a string.
- `element[1]` = the ODBC diagnostic state code, as a string.
- `element[2]` = the ODBC native error code, as a number.

There can be more than one diagnostic message returned by a single SQL call.

The `Result` and `KeyResult` contain the column and row definitions resulting from an SQL command. This definition is shared with the single-threaded ODBC implementation. The complete definition for `ODBCResult`is:

```
class ODBCResult
{
    columns;
    rows;
}
```

The `columns` member is an array of instances of the `ODBCColumn` class:

```
class ODBCColumn
{
    columnsize;     // A numeric column width.
    datatype;       // A number representing the ODBC data type.
    decimaldigits;  // The number of decimal digits, or 0.
    name;           // The column name.
    nullable;       // 0 if not nullable, 1 if nullable.
}
```

The `rows` member is an array of values, each of which corresponds to the column definition in the same position in the `columns` array.

SQLGUID

`SQLGUID` — holds ID strings.

Synopsis

```
class SQLGUID
{
}
```

Description

This structure is used to hold ID strings. For more information, please refer to C Interval Structure.

SQL_DAY_SECOND_STRUCT

SQL_DAY_SECOND_STRUCT — contains time data for SQL_INTERVAL_STRUCT.

Synopsis

```
class SQL_DAY_SECOND_STRUCT
{
    day;
    fraction;
    hour;
    minute;
    second;
}
```

Description

This structure contains time data for SQL_INTERVAL_STRUCT. For more information, please refer to C Interval Structure.

SQL_INTERVAL_STRUCT

SQL_INTERVAL_STRUCT — contains interval data for SQL queries.

Synopsis

```
class SQL_INTERVAL_STRUCT
{
    interval_sign;
    interval_type;
}
```

Description

This structure contains interval data for SQL queries. For more information, please refer to C Interval Structure.

SQL_INTERVAL_STRUCT_intval

SQL_INTERVAL_STRUCT_intval — contains year/month or day/second info for SQL_INTERVAL_STRUCT.

Synopsis

```
class SQL_INTERVAL_STRUCT_intval
{
    day_second;
    year_month;
}
```

Description

This structure contains year/month or day/second info for SQL_INTERVAL_STRUCT. For more information, please refer to C Interval Structure.

SQL_NUMERIC_STRUCT

SQL_NUMERIC_STRUCT — specifies number precision and sign.

Synopsis

```
class SQL_NUMERIC_STRUCT
{
    precision;
    sign;
}
```

Description

This structure specifies number precision and sign. For more information, please refer to C Data Types.

SQL_YEAR_MONTH_STRUCT

SQL_YEAR_MONTH_STRUCT — contains year and month data for SQL_INTERVAL_STRUCT.

Synopsis

```
class SQL_YEAR_MONTH_STRUCT
{
    month;
    year;
}
```

Description

This structure contains year and month data for SQL_INTERVAL_STRUCT. For more information, please refer to C Interval Structure.

TIMESTAMP_STRUCT

TIMESTAMP_STRUCT — contains timestamp data (y,m,d,h,m,s, etc.).

Synopsis

```
class TIMESTAMP_STRUCT
{
    day;
    fraction;
    hour;
    minute;
    month;
    second;
    year;
}
```

Description

This structure contains timestamp data. For more information, please refer to C Data Types.

TIME_STRUCT

TIME_STRUCT — contains time data (h,m,s).

Synopsis

```
class TIME_STRUCT
{
    hour;
    minute;
    second;
}
```

Description

This structure contains time data. For more information, please refer to C Data Types.

Chapter 6. Global Functions

ODBC_AllocEnvironment

`ODBC_AllocEnvironment` — creates an `ODBCEnvironment`.

Synopsis

```
ODBC_AllocEnvironment ()
```

Description

This function is used to create an `ODBCEnvironment` class, the first step in creating an ODBC database. When you allocate the ODBC environment in your script, you can optionally specify the ODBC version that you want. To do this, give the version number (2 or 3) to `ODBC_AllocEnvironment`, like this: `ODBC_AllocEnvironment(2);`

ODBC_ValueString

ODBC_ValueString —

Synopsis

```
ODBC_ValueString (value)
```

Description

Not yet documented.

Chapter 7. Constants

ODBCVER
SQL_ACCESSIBLE_PROCEDURES
SQL_ACCESSIBLE_TABLES
SQL_ALL_TYPES
SQL_ALTER_TABLE
SQL_AM_CONNECTION
SQL_AM_NONE
SQL_AM_STATEMENT
SQL_API_SQLALLOCCONNECT
SQL_API_SQLALLOCENV
SQL_API_SQLALLOCHANDLE
SQL_API_SQLALLOCSTMT
SQL_API_SQLBINDCOL
SQL_API_SQLBINDPARAM
SQL_API_SQLCANCEL
SQL_API_SQLCLOSECURSOR
SQL_API_SQLCOLATTRIBUTE
SQL_API_SQLCOLUMNS
SQL_API_SQLCONNECT
SQL_API_SQLCOPYDESC
SQL_API_SQLDATASOURCES
SQL_API_SQLDESCRIBECOL
SQL_API_SQLDISCONNECT
SQL_API_SQLENDTRAN
SQL_API_SQLERROR
SQL_API_SQLEXECDIRECT
SQL_API_SQLEXECUTE
SQL_API_SQLFETCH
SQL_API_SQLFETCHSCROLL
SQL_API_SQLFREECONNECT
SQL_API_SQLFREEENV
SQL_API_SQLFREEHANDLE
SQL_API_SQLFREESTMT
SQL_API_SQLGETCONNECTATTR
SQL_API_SQLGETCONNECTOPTION
SQL_API_SQLGETCURSORNAME
SQL_API_SQLGETDATA
SQL_API_SQLGETDESCFIELD
SQL_API_SQLGETDESCREC
SQL_API_SQLGETDIAGFIELD
SQL_API_SQLGETDIAGREC
SQL_API_SQLGETENVATTR
SQL_API_SQLGETFUNCTIONS
SQL_API_SQLGETINFO
SQL_API_SQLGETSTMTATTR
SQL_API_SQLGETSTMTOPTION
SQL_API_SQLGETTYPEINFO
SQL_API_SQLNUMRESULTCOLS
SQL_API_SQLPARAMDATA
SQL_API_SQLPREPARE
SQL_API_SQLPUTDATA
SQL_API_SQLROWCOUNT
SQL_API_SQLSETCONNECTATTR
SQL_API_SQLSETCONNECTOPTION
SQL_API_SQLSETCURSORNAME
SQL_API_SQLSETDESCFIELD
SQL_API_SQLSETDESCREC
SQL_API_SQLSETENVATTR
SQL_API_SQLSETPARAM
SQL_API_SQLSETSTMTATTR
SQL_API_SQLSETSTMTOPTION
SQL_DIAG_SERVER_NAME
SQL_DIAG_SQLSTATE
SQL_DIAG_SUBCLASS_ORIGIN
SQL_DIAG_UNKNOWN_STATEMENT
SQL_DIAG_UPDATE_WHERE
SQL_DOUBLE
SQL_DROP
SQL_ERROR
SQL_FALSE
SQL_FETCH_ABSOLUTE
SQL_FETCH_DIRECTION
SQL_FETCH_FIRST
SQL_FETCH_LAST
SQL_FETCH_NEXT
SQL_FETCH_PRIOR
SQL_FETCH_RELATIVE
SQL_FLOAT
SQL_GETDATA_EXTENSIONS
SQL_HANDLE_DBC
SQL_HANDLE_DESC
SQL_HANDLE_ENV
SQL_HANDLE_STMT
SQL_IC_LOWER
SQL_IC_MIXED
SQL_IC_SENSITIVE
SQL_IC_UPPER
SQL_IDENTIFIER_CASE
SQL_IDENTIFIER_QUOTE_CHAR
SQL_INDEX_ALL
SQL_INDEX_CLUSTERED
SQL_INDEX_HASHED
SQL_INDEX_OTHER
SQL_INDEX_UNIQUE
SQL_INSENSITIVE
SQL_INTEGER
SQL_INTEGRITY
SQL_INVALID_HANDLE
SQL_IS_DAY
SQL_IS_DAY_TO_HOUR
SQL_IS_DAY_TO_MINUTE
SQL_IS_DAY_TO_SECOND
SQL_IS_HOUR
SQL_IS_HOUR_TO_MINUTE
SQL_IS_HOUR_TO_SECOND
SQL_IS_MINUTE
SQL_IS_MINUTE_TO_SECOND
SQL_IS_MONTH
SQL_IS_SECOND
SQL_IS_YEAR
SQL_IS_YEAR_TO_MONTH
SQL_MAXIMUM_CATALOG_NAME_LENGTH
SQL_MAXIMUM_COLUMNS_IN_GROUP_BY
SQL_MAXIMUM_COLUMNS_IN_INDEX
SQL_MAXIMUM_COLUMNS_IN_ORDER_BY
SQL_MAXIMUM_COLUMNS_IN_SELECT
SQL_MAXIMUM_COLUMN_NAME_LENGTH
SQL_MAXIMUM_CONCURRENT_ACTIVITIES
SQL_MAXIMUM_CURSOR_NAME_LENGTH
SQL_MAXIMUM_DRIVER_CONNECTIONS
SQL_MAXIMUM_IDENTIFIER_LENGTH
SQL_MAXIMUM_INDEX_SIZE

```
SQL_API_SQLSPECIALCOLUMNS
SQL_API_SQLSTATISTICS
SQL_API_SQLTABLES
SQL_API_SQLTRANSACT
SQL_ARD_TYPE
SQL_ATTR_APP_PARAM_DESC
SQL_ATTR_APP_ROW_DESC
SQL_ATTR_AUTO_IPD
SQL_ATTR_CURSOR_SCROLLABLE
SQL_ATTR_CURSOR_SENSITIVITY
SQL_ATTR_IMP_PARAM_DESC
SQL_ATTR_IMP_ROW_DESC
SQL_ATTR_METADATA_ID
SQL_ATTR_OUTPUT_NTS
SQL_CATALOG_NAME
SQL_CB_CLOSE
SQL_CB_DELETE
SQL_CB_PRESERVE
SQL_CHAR
SQL_CLOSE
SQL_CODE_DATE
SQL_CODE_TIME
SQL_CODE_TIMESTAMP
SQL_COLLATION_SEQ
SQL_COMMIT
SQL_CURSOR_COMMIT_BEHAVIOR
SQL_CURSOR_SENSITIVITY
SQL_DATA_AT_EXEC
SQL_DATA_SOURCE_NAME
SQL_DATA_SOURCE_READ_ONLY
SQL_DATETIME
SQL_DATE_LEN
SQL_DBMS_NAME
SQL_DBMS_VER
SQL_DECIMAL
SQL_DEFAULT
SQL_DEFAULT_TXN_ISOLATION
SQL_DESCRIBE_PARAMETER
SQL_DESC_ALLOC_AUTO
SQL_DESC_ALLOC_TYPE
SQL_DESC_ALLOC_USER
SQL_DESC_COUNT
SQL_DESC_DATA_PTR
SQL_DESC_DATETIME_INTERVAL_CODE
SQL_DESC_INDICATOR_PTR
SQL_DESC_LENGTH
SQL_DESC_NAME
SQL_DESC_NULLABLE
SQL_DESC_OCTET_LENGTH
SQL_DESC_OCTET_LENGTH_PTR
SQL_DESC_PRECISION
SQL_DESC_SCALE
SQL_DESC_TYPE
SQL_DESC_UNNAMED
SQL_DIAG_ALTER_DOMAIN
SQL_DIAG_ALTER_TABLE
SQL_DIAG_CALL
SQL_DIAG_CLASS_ORIGIN
SQL_DIAG_CONNECTION_NAME
SQL_DIAG_CREATE_ASSERTION
SQL_DIAG_CREATE_CHARACTER_SET
SQL_DIAG_CREATE_COLLATION
SQL_DIAG_CREATE_DOMAIN
SQL_DIAG_CREATE_INDEX
SQL_MAXIMUM_ROW_SIZE
SQL_MAXIMUM_SCHEMA_NAME_LENGTH
SQL_MAXIMUM_STATEMENT_LENGTH
SQL_MAXIMUM_TABLES_IN_SELECT
SQL_MAXIMUM_USER_NAME_LENGTH
SQL_MAX_CATALOG_NAME_LEN
SQL_MAX_COLUMNS_IN_GROUP_BY
SQL_MAX_COLUMNS_IN_INDEX
SQL_MAX_COLUMNS_IN_ORDER_BY
SQL_MAX_COLUMNS_IN_SELECT
SQL_MAX_COLUMNS_IN_TABLE
SQL_MAX_COLUMN_NAME_LEN
SQL_MAX_CONCURRENT_ACTIVITIES
SQL_MAX_CURSOR_NAME_LEN
SQL_MAX_DRIVER_CONNECTIONS
SQL_MAX_IDENTIFIER_LEN
SQL_MAX_INDEX_SIZE
SQL_MAX_MESSAGE_LENGTH
SQL_MAX_NUMERIC_LEN
SQL_MAX_ROW_SIZE
SQL_MAX_SCHEMA_NAME_LEN
SQL_MAX_STATEMENT_LEN
SQL_MAX_TABLES_IN_SELECT
SQL_MAX_TABLE_NAME_LEN
SQL_MAX_USER_NAME_LEN
SQL_NAMED
SQL_NC_HIGH
SQL_NC_LOW
SQL_NEED_DATA
SQL_NONSCROLLABLE
SQL_NO_DATA
SQL_NO_NULLS
SQL_NTS
SQL_NULLABLE
SQL_NULLABLE_UNKNOWN
SQL_NULL_COLLATION
SQL_NULL_DATA
SQL_NULL_HDBC
SQL_NULL_HDESC
SQL_NULL_HENV
SQL_NULL_HSTMT
SQL_NUMERIC
SQL_OJ_CAPABILITIES
SQL_ORDER_BY_COLUMNS_IN_SELECT
SQL_OUTER_JOIN_CAPABILITIES
SQL_PC_NON_PSEUDO
SQL_PC_PSEUDO
SQL_PC_UNKNOWN
SQL_PRED_BASIC
SQL_PRED_CHAR
SQL_PRED_NONE
SQL_REAL
SQL_RESET_PARAMS
SQL_ROLLBACK
SQL_ROW_IDENTIFIER
SQL_SCOPE_CURROW
SQL_SCOPE_SESSION
SQL_SCOPE_TRANSACTION
SQL_SCROLLABLE
SQL_SCROLL_CONCURRENCY
SQL_SEARCH_PATTERN_ESCAPE
SQL_SENSITIVE
SQL_SERVER_NAME
SQL_SMALLINT
```

```
SQL_DIAG_CREATE_SCHEMA
SQL_DIAG_CREATE_TABLE
SQL_DIAG_CREATE_TRANSLATION
SQL_DIAG_CREATE_VIEW
SQL_DIAG_DELETE_WHERE
SQL_DIAG_DROP_ASSERTION
SQL_DIAG_DROP_CHARACTER_SET
SQL_DIAG_DROP_COLLATION
SQL_DIAG_DROP_DOMAIN
SQL_DIAG_DROP_INDEX
SQL_DIAG_DROP_SCHEMA
SQL_DIAG_DROP_TABLE
SQL_DIAG_DROP_TRANSLATION
SQL_DIAG_DROP_VIEW
SQL_DIAG_DYNAMIC_DELETE_CURSOR
SQL_DIAG_DYNAMIC_FUNCTION
SQL_DIAG_DYNAMIC_FUNCTION_CODE
SQL_DIAG_DYNAMIC_UPDATE_CURSOR
SQL_DIAG_GRANT
SQL_DIAG_INSERT
SQL_DIAG_MESSAGE_TEXT
SQL_DIAG_NATIVE
SQL_DIAG_NUMBER
SQL_DIAG_RETURNCODE
SQL_DIAG_REVOKE
SQL_DIAG_ROW_COUNT
SQL_DIAG_SELECT_CURSOR
SQL_SPECIAL_CHARACTERS
SQL_STILL_EXECUTING
SQL_SUCCESS
SQL_SUCCESS_WITH_INFO
SQL_TC_ALL
SQL_TC_DDL_COMMIT
SQL_TC_DDL_IGNORE
SQL_TC_DML
SQL_TC_NONE
SQL_TIMESTAMP_LEN
SQL_TIME_LEN
SQL_TRANSACTION_CAPABLE
SQL_TRANSACTION_ISOLATION_OPTION
SQL_TRUE
SQL_TXN_CAPABLE
SQL_TXN_ISOLATION_OPTION
SQL_TYPE_DATE
SQL_TYPE_TIME
SQL_TYPE_TIMESTAMP
SQL_UNBIND
SQL_UNKNOWN_TYPE
SQL_UNNAMED
SQL_UNSPECIFIED
SQL_USER_NAME
SQL_VARCHAR
SQL_XOPEN_CLI_YEAR
```

Index

G

H

I

M

N

O

P

Q

R

S

T

U

V

Colophon

This book was produced by Cogent Real-Time Systems, Inc. from a single-source group of SGML files. Gnu Emacs was used to edit the SGML files. The DocBook DTD and related DSSSL stylesheets were used to transform the SGML source into HTML, PDF, and QNX Helpviewer output formats. This processing was accomplished with the help of OpenJade, JadeTeX, Tex, and various scripts and makefiles. Details of the process are described in our book: Preparing Cogent Documentation, which is published on-line at
`http://developers.cogentrts.com/cogent/prepdoc/book1.html`.

Text written by Bob McIlvride and Andrew Thomas.