insightsoftware

Simba SDK

Build a C++ ODBC Driver for NON SQL-Capable Data Sources in 5 Days

Version 10.4

November 2026

Copyright

This document was released in November 2026.

Copyright ©2014–2026 insightsoftware. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission from insightsoftware.

The information in this document is subject to change without notice. insightsoftware strives to keep this information accurate but does not warrant that this document is error-free.

Any insightsoftware product described herein is licensed exclusively subject to the conditions set forth in your insightsoftware license agreement.

Simba, the Simba logo, SimbaEngine, and Simba Technologies are registered trademarks of Simba Technologies Inc. in Canada, the United States and/or other countries. All other trademarks and/or servicemarks are the property of their respective owners.

All other company and product names mentioned herein are used for identification purposes only and may be trademarks or registered trademarks of their respective owners.

Information about the third-party products is contained in a third-party-licenses.txt file that is packaged with the software.

Contact Us

www.simba.com

Contents

Copyright **2**

Contents **3**

About this Guide **6**

Purpose 6

Advantages of Using the Simba SDK 6

Audience 7

Prerequisites 7

Supported Compilers and Platforms 8

Document Conventions 8

Variables Used in this Document 9

Product Overview **11**

ODBC Standards 11

The Simba SDK Solution 11

About the QuickStart Sample Connector 13

Day One (Windows): Environment Setup **15**

Installing Simba SDK 15

Platform-Specific Package Names 16

Building the QuickStart Sample Driver 16

Examining Windows Registry Keys 17

Viewing DSNs Using ODBC Data Source Administrator 18

Connecting to the Data Source 19

Setting Up a Customizable Project 19

Building Your Customizable Project 20
Updating the Windows Registry 21
Viewing Your DSNs Using ODBC Data Source Administrator 22
Connecting to the Data Source Using Your Customizable Driver 22
Summary – Day One (Windows) 23
Day One (Linux/Unix and macOS): Environment Setup 25
Install the Simba SDK 25
Build the Sample ODBC Connector 27
Configure the Connector and Data Source 29
Connect to the Data Source 34
Set up a Custom ODBC Connector Project 36
Configure Your Custom Connector and Data Source 37
Debug Your Custom Connector 40
Enable Logging 41
Summary – Day One (Linux/Unix and macOS) 42
Day Two: Connection Management 44
Construct the Connector Singleton (Set the Configuration Branding) 44
Set Connector Properties 45
Set Logging Details 46
Check Connection Settings 46
Establish a Connection 48
Summary – Day Two 49
Day Three: Metadata Implementation 50

Create and Return Metadata Sources 50

Summary – Day Three 54

Day Four: Data Retrieval 55

Enable Data Retrieval 55

Summary – Day Four 58

Day Five: Productization 60

Rebrand Error Messages 60

Update the vendor name in the Tableau Datasource Connection (TDC) file 61

Rebrand the Custom ODBC Connector (Adding Finishing Touches) 61

Summary – Day Five 62

Conclusion 63

Reference 64

Appendix A: ODBC Data Source Administrator on Windows 32-Bit vs. 64-Bit 64

Appendix B: Bitness and the Windows Registry 65

Appendix C: Driver Managers (Linux/Unix and macOS) 66

Appendix D: Locating the Configuration Files (Linux/Unix and macOS) 66

Appendix E: Data Retrieval 69

Appendix F: How to Add Schema Support 73

Appendix G: Install the Evaluation License 74

Appendix H: Troubleshooting (Linux/Unix and macOS) 74

Appendix J: Third-Party Licenses 75

Contact Us 79

Third-Party Trademarks 80

About this Guide

Purpose

This guide explains how to use the Simba SDK to create a custom ODBC connector for a data store that is not SQL-capable, on Windows, Linux/Unix, and macOS platforms. It explains how to customize the QuickStart sample connector, which is included in Simba SDK.

Using this sample connector is the quickest and easiest way to create a custom ODBC connector. At the end of five days, you will have a read-only connector that connects to your data store.

Note:
An online version of this guide is located at Simba SDK - Driver Development.

Advantages of Using the Simba SDK

The ODBC specification defines a rich interface that allows any ODBC-enabled application to connect to a data store. To implement a connector that supports this specification, developers must understand all the complexities of error checking, session management, and data conversion, then design their code in a robust and efficient manner. Developers must also understand how to optimize data retrieval to get maximum performance when connecting to large and complex data stores.

For data stores that do not support SQL, the SDK provides an SQL parser and an execution engine. Developers can use these features to translate SQL queries to a custom API that the data store understands.

The Simba SDK product, developed by experts in the field, is a complete implementation of the ODBC specification. It contains an easy-to-use SDK that allows you to create a robust and efficient connector for your data store.

Build a Custom ODBC Connector in Five Days

Over the course of five days, this guide explains how to accomplish the following tasks:

1. Set up the development environment and build the sample connector.
2. Use the sample connector as a template to create a custom ODBC connector.
3. Make a connection to the data store.
4. Retrieve metadata.
5. Retrieve data.
6. Rename and rebrand the custom ODBC connector.

In the QuickStart connector, the areas of code that require modification are marked with “TODO” messages and a short explanation. Some of these changes customize the connector for your specific data store, while other changes rename the connector for your company or product.

SEN SDK 10.4:

This guide reflects the SEN SDK 10.4 API. Code examples that changed between 10.3 and 10.4 are flagged with a "SEN SDK 10.4:" note where they appear.

Audience

The guide is intended for developers who want to use Simba SDK to build a connector for a data store that is not SQL-capable.

Prerequisites

Before beginning this tutorial, ensure you have the following. Requirements are listed by platform; review the section for your development platform, then the common requirements that apply to all platforms.

Windows Requirements

Requirement	Details
Development Environment	Microsoft Visual Studio 2022 or Visual Studio 2026 (Community, Professional, or Enterprise)
Operating System	Windows 11 or Windows Server 2016/2019/2022/2025
Testing Tools	Microsoft Data Access SDK 2.8 (includes the ODBC Test tool)
Disk Space	Minimum 2 GB free disk space for the SDK and development files

Linux Requirements

Requirement	Details
Development Environment	GCC 8.5 GNU Make 4; CMake 3.17.0
Operating System	Ubuntu 20.04 LTS or later, RHEL 8 or later, or an equivalent Linux distribution
Driver Manager	unixODBC 2.3.x or iODBC 3.52.x
Disk Space	Minimum 2 GB free disk space for the SDK and development files

macOS Requirements

Requirement	Details
Development Environment	Xcode 26, 16, or 15 with the Command Line Tools installed
Operating System	macOS Tahoe 26, macOS Sonoma 14.5+, or macOS Ventura 13.5+
Driver Manager	iODBC or unixODBC Driver Manager installed
Disk Space	Minimum 2 GB free disk space for the SDK and development files

Common Requirements (All Platforms)

Requirement	Details
Programming Skills	Proficiency in C++ programming (intermediate level or higher)

Requirement	Details
Knowledge	Basic understanding of ODBC concepts, SQL syntax, and database fundamentals
Software	Simba SDK Version 10.4 (licensed)

Important: You must have a valid Simba SDK license before beginning this tutorial. Contact insightsoftware sales at www.insightsoftware.com if you need to obtain a license.

Supported Compilers and Platforms

Windows – Supported Visual Studio Versions

Simba SDK version 10.4 supports the following Visual Studio versions on Windows:

Visual Studio Version	Release Date	Status
2026	February 2026	Current (Recommended)
2022	November 2021	Supported

Important: ARM64 builds are supported only on Windows 11.

macOS – Supported Xcode Versions

Simba SDK version 10.4 supports the following Xcode versions on macOS:

Xcode Version	Release Date	macOS Requirement	Status
Xcode 26.x	September 2025	macOS Tahoe 26 or later	Current (Recommended)
Xcode 16.x	September 2024	macOS Sonoma 14.5 or later	Supported
Xcode 15.x	September 2023	macOS Ventura 13.5 or later	Supported (Legacy)

To check your installed Xcode version, open Terminal and type: xcodebuild -version

Linux – Supported Compilers

Simba SDK version 10.4 supports the following compilers on Linux:

Compiler	Minimum Version	Recommended Version	Status
GCC (g++)	8.5	8.5	Recommended

To check your installed GCC version, open a terminal and type: g++ --version

Document Conventions

This guide uses the following typographical and formatting conventions:

Convention	Meaning	Example
Bold text	User interface elements a user clicks and text a	Click Build > Build Solution

Convention	Meaning	Example
	user types	
Monospace font	Code, file names, paths, and commands	QuickStart_vs2022.sln
[INSTALL_DIR]	Simba SDK installation directory	C:\Simba Technologies
<YourProjectName>	Placeholder for your custom project name	<YourProjectName>.vcxproj
Italics	Emphasis or document titles	Simba SDK - Driver Development

Platform-specific instructions are identified with a shaded heading, for example Windows, Linux/Unix, macOS, or Linux and macOS. Follow the instructions for your development platform.

Each day chapter begins with a LEARNING OBJECTIVES box listing what you will be able to do by the end of the day and ends with a ✓ Day Checkpoint box that you can use to verify your progress, followed by a summary of the completed tasks.

Note: Indicates a short note appended to a paragraph.

Important: Indicates an important comment related to the preceding paragraph.

Tutorial Structure

This tutorial is organized into five progressive days:

Day	Focus Area	Key Topics	Est. Time
Day 1	Environment Setup	Installation, building the sample connector, registry/DSN setup, project setup	4-6 hours
Day 2	Connection Management	Connector singleton, properties, logging, connections	4-6 hours
Day 3	Metadata Implementation	Catalog functions, type info, tables, columns	4-6 hours
Day 4	Data Retrieval	Opening tables, column metadata, retrieving data	4-6 hours
Day 5	Productization	Error handling, branding, dialogs, finalization	4-6 hours

Variables Used in this Document

The following variables are used in this document:

Variable	Description
[INSTALL_DIR]	Installation directory for the Simba SDK. Default value on Linux, Unix, and macOS platforms: [UNTAR_DIR]/SimbaEngineSDK/10.4. On Windows, the default installation directory (also referred to as InstallDir or [INSTALL_DIR]) is C:\Simba Technologies.
[UNTAR_DIR]	Directory where the Simba SDK distributable was untarred (Linux, Unix, and

Variable	Description
	macOS platforms).
<YourProjectName>	The name you choose for your custom connector project (Windows platforms).

Product Overview

Applications, such as PowerBI and Tableau, use connectors to connect to data stores from which they read and write data. Applications support the ODBC protocol to enable connection with any connector that also supports ODBC. A connector exposes the ODBC protocol to the application and another API, such as SQL or a custom API, to the data store.

Note:

This guide explains how to create an ODBC connector for a data store that is not SQL-capable. To create an ODBC connector for a data store that is SQL-capable, see Simba SDK - Driver Development.

ODBC Standards

ODBC is one of the most established and widely supported APIs for connecting to and working with databases. A main component of this technology is the ODBC connector, which connects an application to the database.

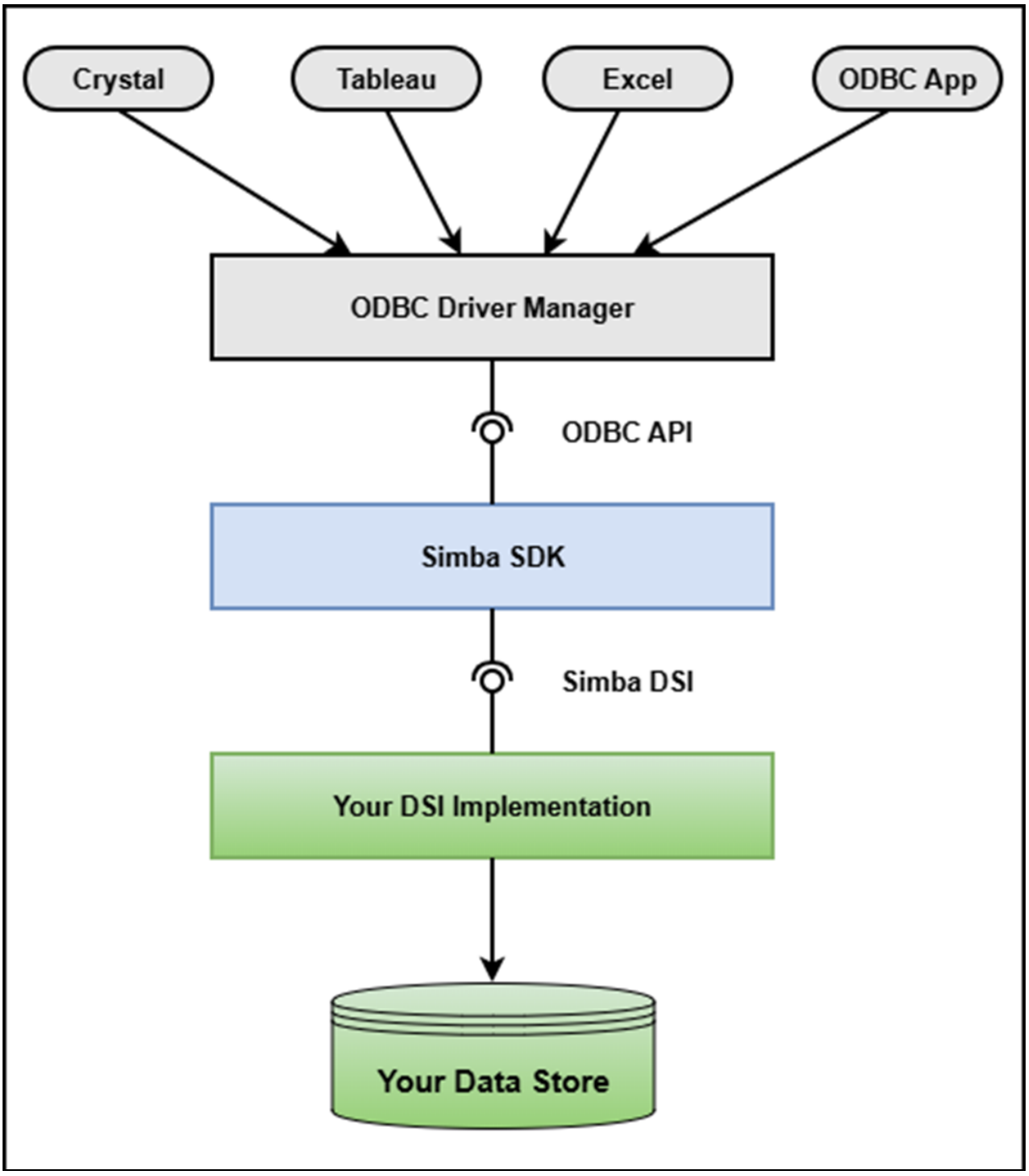
For a brief description of the ODBC standard, see https://www.insightsoftware.com/simba/blog/what-is-odbc/.

For complete information on the ODBC 3.80 specification, see the ODBC Programmer's Reference at http://msdn.microsoft.com/en-us/library/ms714177(v=vs.85).aspx.

The Simba SDK Solution

Connectors based on the Simba SDK leverage its error checking, session management, data conversion, optimization, and other low-level implementation details. The Simba SDK uses ODBC to communicate with the driver manager and a simple API (called the Data Store Interface API or DSI API) to communicate with the data store. The DSI API defines the primitive operations needed to access a data store.

The figure below shows a typical ODBC stack:



A typical ODBC stack

SDK developers create an implementation of a DSI (also known as a DSI Implementation or DSII) that applications use to access the target data store in the process of executing an SQL statement. In the final executable, the components from Simba SDK take responsibility for meeting the data access standards while the custom DSI implementation takes responsibility for accessing the data store and translating it to the DSI API.

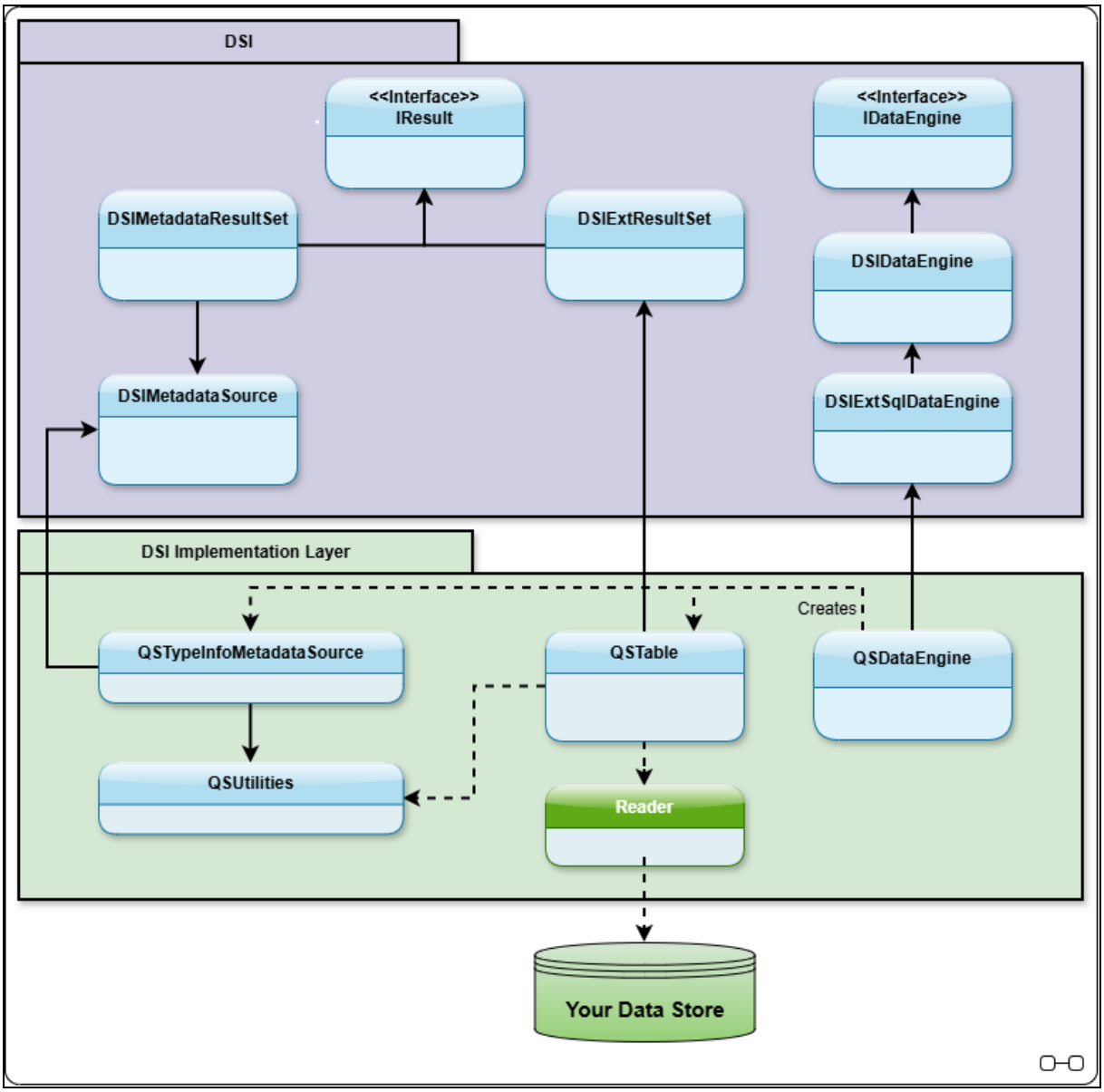
The executable file can be a shared library or stand-alone server executable. ODBC applications, such as Tableau or Microsoft Excel, use this executable when connecting to the data store in the process of executing an SQL statement.

About the QuickStart Sample Connector

Simba SDK includes a sample connector that you can use as a template to create a custom ODBC connector for data stores that are not SQL-capable. The QuickStart connector is a sample DSI API implementation of an ODBC connector, written in C++, that reads files in tab-separated Unicode text format. The Simba SQLEngine is used to perform SQL processing, allowing the QuickStart connector to retrieve data from the sample text files, which are not a SQL-capable data source.

Using the QuickStart sample connector to prototype a DSI implementation for a custom data store helps developers understand howSimba SDK works. By removing the shortcuts and simplifications implemented in the QuickStart connector, you can use it as the foundation for a commercial DSI implementation and create a custom ODBC connector for a data store that is not SQL-capable.

The UML diagram below shows a typical design pattern for a DSI implementation:



Design pattern for a DSI implementation

Notice the circular pattern of class relationships, headed by IResult and anchored by QSUtilities. The IResult class is responsible for retrieving column data and maintaining a cursor across result rows. The QSUtilities class contains platform-specific utility functions.

To implement data retrieval, your Reader class interacts directly with your data store to retrieve data and then deliver the data to the QSTable class on demand. The Reader class should manage caching, buffering, paging and all the other techniques to speed data access.

As a starting point, to make your connector work properly with Microsoft Excel you can add metadata access by implementing the QSTypeInfoMetadataSource class and using the DSIExtMetadataHelper class. The DSIExtMetadataHelper class is responsible for iterating through tables and stored procedures so the engine can generate catalog function metadata.

Day One (Windows): Environment Setup

Today's task is to set up and test the development environment and project files for the driver. By the end of the day, you will have compiled and tested your first ODBC driver.

LEARNING OBJECTIVES

By the end of this section, you will be able to:

- Install and configure the Simba SDK on Windows
- Build the QuickStart sample driver using Visual Studio
- Understand ODBC registry key configuration
- View and configure data sources in the ODBC Data Source Administrator
- Test ODBC connectivity using the ODBC Test tool
- Set up a new project for your custom ODBC driver

The Day One instructions for Windows explain how to set up and prepare your development environment to create a driver based on the QuickStart sample driver:

1. Install Simba SDK.
2. Build the QuickStart sample driver.
3. Review the keys that the QuickStart sample driver adds to the Windows Registry.
4. In the ODBC Data Source Administrator, review the DSNs that the QuickStart sample driver creates.
5. Using the QuickStart sample driver, read the tabbed Unicode text format sample data.
6. Create a customizable driver project based on the QuickStart sample driver.
7. Test your customizable driver project by reading the tabbed Unicode text format QuickStart sample data.

Detailed instructions for each step appear below.

Installing Simba SDK

To install Simba SDK:

1. Uninstall any previous versions of Simba SDK installed on your computer.
2. Close Visual Studio.
3. Double-click to run the Simba SDK setup executable that corresponds to your version of Visual Studio, and then follow the instructions provided in the installation wizard.

Platform-Specific Package Names

Platform	Example Package Name
Windows Intel (vs 2022)	SimbaEngineSDK_Release_Windows_vs2022_10.4.*
Windows Intel (vs 2026)	SimbaEngineSDK_Release_Windows_vs2026_10.4.*
Windows ARM (vs 2022)	SimbaEngineSDK_Release_Windows_ARM64_vs2022_10.4.*
Windows ARM (vs 2026)	SimbaEngineSDK_Release_Windows_ARM64_vs2026_10.4.*

Note:
ARM is only supported on Windows 11.

Important:
The Simba SDK environment variables are defined only for the user that ran the installation. If you install Simba SDK as a regular user and then run Visual Studio as an administrator, SimbaEngine will not work properly.

In this guide, the directory where you install Simba SDK is referred to using the placeholder InstallDir. The default installation directory is `C:\insightsoftware`. When working through procedures, ensure that you replace the [INSTALL_DIR] placeholder with the appropriate file path.

Building the QuickStart Sample Driver

To build the Quickstart sample driver:

1. In Microsoft Visual Studio 2022, click **File** > **Open**> **Project** or **Solution**.

   Note: Note: Visual Studio 2022 is used as an example, but other versions are also supported.

2. In the **Open Project** menu, navigate to the folder `InstallDir\SimbaEngineSDK\Examples\Source\Quickstart\Source`, then select the file named *`QuickstartDSII_.vcxproj`*, and then click **Open**.
3. Click **Build** > **Configuration Manager**. Click the drop-down arrow next to the **Active Solution Configuration** field, then select **Debug_MTDLL**, and then click **Close**.
4. Click **Build** > **Build Solution** or press **F7**.

For 32-bit drivers, the build appears in the folder `InstallDir\SimbaEngineSDK\Examples\Source\Quickstart\Bin\Win32\Debug_MTDLL`.

For 64-bit drivers, the build appears in the folder `InstallDir\SimbaEngineSDK\Examples\Source\Quickstart\Bin\x64\Debug_MTDLL (Intel)` and `InstallDir\SimbaEngineSDK\Examples\Source\Quickstart\Bin\ARM64\Debug_MTDLL (ARM)`.

Examining Windows Registry Keys

Installing Simba SDK adds or updates keys in the Windows Registry to define the sample drivers included with Simba SDK, as well as Data Source Names (DSNs) using the sample drivers.

The Simba SDK installer automatically added or updated the following registry keys that define Data Source Names (DSNs) and driver locations:

Registry Key	Purpose
ODBC Data Sources	Lists each DSN or driver pair.
QuickstartDSII	Defines the Data Source Name (DSN). Used by the ODBC Driver Manager to connect your driver to your database.
ODBC Drivers	Lists the drivers that are installed.
QuickstartDSIIDriver	Defines the driver and its setup location. The ODBC Driver Manager uses this key to connect to and configure your driver.

To view the Registry keys related to the QuickStart sample driver:

1. Click **Start**, then type **regedit.exe** in the **Search Programs** and **Files** field, and then press **ENTER**.
   - In the **Registry Editor**, to view the key defining the 32-bit driver on 32-bit Windows or the 64-bit driver on 64-bit Windows, browse to ***HKEY_LOCAL_MACHINE/SOFTWARE/ODBC/ODBCINST.INI/QuickstartDSIIDriver*** and ***HKEY_LOCAL_MACHINE/SOFTWARE/ODBC/ODBCINST.INI/ODBC Drivers*** OR
   - To view the key defining the 32-bit driver on 64-bit Windows, browse to ***HKEY_LOCAL_MACHINE/SOFTWARE/WOW6432NODE/ODBC/ODBCINST.INI/ QuickstartDSIIDriver*** and ***HKEY_LOCAL_MACHINE/SOFTWARE/WOW6432NODE/ODBC/ODBCINST.INI/ODBC Drivers***.
2. To view the key defining the DSN:
   - Using the 32-bit driver on 32-bit Windows or the 64-bit driver on 64-bit Windows, browse to ***HKEY_LOCAL_MACHINE/SOFTWARE/ODBC/ ODBC.INI/QuickstartDSII*** OR
   - Using the 32-bit driver on 64-bit Windows, browse ***HKEY_LOCAL_MACHINE/SOFTWARE/WOW6432NODE/ODBC/ODBC.INI/ QuickstartDSII***.
3. Observe the following values set in the **QuickstartDSII** Driver subkey for the 32-bit driver:
   - **Driver**: `[INSTALL_DIR]\Examples\Builds\Bin\Win32\Release_MTDLL\QuickstartDSII_MTDLL.dll`
   - **Setup**: `InstallDir\Examples\Builds\Bin\Win32\Release_MTDLL\QuickstartDSII_MTDLL.dll`
   - **Description**: `Sample 32-bit Simba SDK Quickstart DSII`
4. Observe the following values set in the **QuickstartDSII** Driver subkey for the 64-bit driver:
   - **Driver**: `InstallDir\Examples\Builds\Bin\x64\Release_MTDLL\QuickstartDSII_MTDLL.dll`

- **Setup**: `InstallDir\Examples\Builds\Bin\x64\Release_MTDLL\QuickstartDSII_MTDLL.dll`
- **Description**: `Sample 64-bit Simba SDK Quickstart DSII`

5. Also in the **ODBCINST.INI** subkey, a string value named **QuickstartDSII** Driver and having the value Installed appears in the ODBC Drivers subkey.
6. Observe the following values set in the **QuickstartDSII** subkey for the 32-bit driver:
   - **Driver**: `QuickStartDSII Driver`
   - **DBF**: `InstallDir\Examples\Databases\Quickstart`
   - **Description**: `Sample 32-bit Simba SDK Quickstart DSII`
7. Observe the following values set in the **QuickstartDSII** subkey for the 64-bit driver:
   - **Driver**: `QuickStartDSII Driver`
   - **DBF**: `InstallDir\Examples\Databases\Quickstart`
   - **Description**: `Sample 64-bit Simba SDK Quickstart DSII`
8. Also, in the **ODBC.INI** subkey, a string value named **QuickstartDSII** having the value **QuickstartDSII** Driver appears in the ODBC Data Sources subkey.

Viewing DSNs Using ODBC Data Source Administrator

Familiarize yourself with the DSNs related to the QuickStart sample driver using the ODBC Data Source Administrator.

To view QuickStart DSNs in the ODBC Data Source Administrator:

- Run the Windows ODBC Data Source Administrator. For 32-bit drivers on 32-bit Windows and 64-bit drivers on 64-bit Windows, open the Control Panel, select **Administrative Tools**, and then select **Data Sources (ODBC)**. If your Control Panel is set to view by category, then Administrative Tools is located under System and Security.
- In the ODBC Data Source Administrator, click the **System DSN** tab.
- Scroll through the list of System Data Sources, select **QuickstartDSII** and then click **Configure**. The Data Source Configuration window opens and displays the data source name, description and the data directory.
- Click **Cancel** to close the Data Source Configuration window.

Connecting to the Data Source

To test connecting to the data source using the QuickStart sample driver, you can use any ODBC application, such as Microsoft Excel, Microsoft Access or ODBC Test. In this section, we will use the ODBC Test tool, which is included in the Microsoft Data Access (MDAC) 2.8 Software Development Kit (SDK) available for download from Microsoft Data Access Components (MDAC) 2.8 Software Development Kit.

To test connecting to the data source using the QuickStart sample driver:

1. Start the ODBC Test tool. By default, the ODBC Test application is installed in the following folder: C:\Program Files (x86)\Microsoft Data Access SDK 2.8\Tools\. Navigate to the folder that corresponds to your machine's architecture (amd64, ia64 or x86) and then click odbcte32.exe to launch the ANSI version or click odbct32w.exe to launch the Unicode version. Recommend using the Unicode version unless you are actively trying to debug an application using the ANSI APIs.

   > **Important:** It is important to run the correct version of the ODBC Test tool for ANSI or Unicode and 32-bit or 64-bit.

2. In the ODBC Test tool, click Conn, and then click Full Connect. The Full Connect window opens.
3. In the Full Connect dialog, select QuickstartDSII data source from the list of data sources, and then click OK. If you do not see your data source in the list, ensure that you are running the version of the ODBC Test tool that corresponds to the version of the data source that you created. For example, if you created a 32-bit data source then you need to use the 32-bit version of the ODBC Test tool.

A new window opens displaying a message that you have successfully connected to the data source.

Setting Up a Customizable Project

Now that you have built the example driver, you are ready to set up a development project to create a custom ODBC driver.

> **Important:** Customizing the QuickStart sample driver project directly is not recommended. At times, referring to the original QuickStart sample driver project may help you debug your custom driver. Also, if you customize the QuickStart sample driver project, then your changes may be lost when you upgrade to a new release of Simba Simba SDK in the future.

To set up a customizable project based on the QuickStart sample driver:

1. In your Windows Explorer window, copy the `[INSTALL_DIR]\SimbaEngineSDK\Examples\Source\Quickstart` directory and paste it to the same location. This will create a new directory called "Quickstart - Copy". Rename the directory to something that is meaningful to you. This will be the top-level directory for your new project and DSI implementation files. For the rest of this tutorial, when you see <YourProjectName> in the instructions, replace this with the name you choose for this directory, which is also the name of your project.
2. Open your new directory then open the Source directory and rename the `QuickstartDSII.vcxproj` file in it to <YourProjectName>.vcxproj, where you replace <YourProjectName> with the name of your project. This will be the project file for your new ODBC driver.

3. Rename the `.sln` file. This new <YourProjectName>.sln file is the solution file for your new ODBC driver.
4. Using a text editor, open the project file (`.vcxproj`) and replace every instance of "QuickstartDSII" in the source code with the name of your new ODBC driver. Then save and close the file.
5. Using a text editor, open the solution file (`.sln`) and replace every instance of "QuickstartDSII" in the source code with the name of your new ODBC driver. In addition, the name of the project file must be updated to match the <YourProjectName>.vcxproj project file that you renamed. Then, save and close the file.

Building Your Customizable Project

To build your customizable ODBC driver project based on the QuickStart sample driver project:

1. Launch Microsoft Visual Studio.
2. Click **File** > **Open** > **Project or Solution.**
3. Navigate to `[INSTALL_DIR]\SimbaEngineSDK\Examples\Source\<YourProjectName>\Source and then open the <YourProjectName>.vcxproj file.`
4. Click **Build** > Configuration Manager and make sure that the active solution configuration is Debug_MTDLL, and then click **Close**.
5. Click **Build** > **Build Solution** or press **F7** to build the driver. This will build the Debug_MTDLL version of the driver and place it in the location `[INSTALL_DIR]\SimbaEngineSDK\Examples\Source\<YourProjectName>\Bin\Win32\Debug_MTDLL`.
6. When you build your new project, "TODO" messages appear in the Output window along with the build information. If the Output window is not displayed automatically, you can open it by selecting **Debug** > **Windows** > **Output**.

```
TODO #1: Construct driver singleton.
TODO #2: Set the driver properties.
TODO #3: Set the driver-wide logging details.
TODO #4: Set the connection-wide logging details.
TODO #5: Check connection settings.
TODO #6: Establish A connection.
TODO #7: Create and return your metadata sources.
TODO #8: Open a table.
TODO #9: Register messages xml file for handling by DSIMessageSource.
TODO #10: Set the vendor name, which will be prepended to error messages.
```

Over the next four days, you will be visiting each "TODO" and modifying the source code.

TODO Items Reference

The following table maps each TODO item to the day of this guide in which it is covered:

TODO #	Description and Day
1	Construct connector singleton – Day 2
2	Set the driver properties – Day 2
3	Set the driver-wide logging details – Day 2
4	Set the connection-wide logging details – Day 2
5	Check Connection Settings – Day 2
6	Establish A Connection – Day 2
7	Create and return your Metadata Sources – Day 3
8	Open A Table – Day 4
9	Register Messages xml file for handling by DSIMessageSource – Day 5
10	Set the vendor name, which will be prepended to error messages – Day 5

Updating the Windows Registry

When you set up a project to create a custom driver based on a sample driver provided with Simba SDK, you need to create keys in the Windows Registry similar to the keys that the sample driver uses to define the driver as well as DSNs for connecting the driver to your data source. To create keys, Simba SDK includes .REG files that you can customize and import into the Registry.

To create Registry keys for your custom driver project based on the QuickStart sample driver:

1. In Microsoft Visual Studio, click **File** > **Open** > **File** and navigate to `[INSTALL_DIR]\SimbaEngineSDK\Examples\Source\<YourProjectName>\Source`.
2. For 32-bit Windows, open `SetupMyQuickstartDSII-32on32.reg`. For a 32-bit ODBC driver on 64-bit Windows, open `SetupMyQuickstartDSII-32on64.reg`. For a 64-bit ODBC driver on 64-bit Windows, open `SetupMyQuickstartDSII-64on64.reg`.
3. In the file, replace [INSTALL_DIR] with the path to the installation directory. In the path, you must enter double backslashes. For example, by default, the samples are installed to `C:\Simba Technologies` so in that case, replace all instances of [INSTALL_DIR] with `C:\\Simba Technologies`.
4. Update the ODBC Data Sources section to add your new data source. Under the `[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\ODBC Data Sources]` section, change "`MyQuickstartDSII`"="`MyQuickstartDSIIDriver`" to the name of your new data source and new driver. For example, "`<YourProjectName>DSII`"="`<YourProjectName>DSIIDriver`".
5. Modify the data source definition for that data source. Change the line that says [HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\MyQuickstartDSII] so that it contains your new data source name. For example, [HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\<YourProjectName>DSII].
6. Beside the line that starts with "Driver"=, enter the path to the driver dll file.
7. Update the ODBC Drivers section to add your new driver. Under the `[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\ODBC Drivers]` section, change

"MyQuickstartDSIIDriver"="Installed" to match the name of your new driver. For example, "<YourProjectName>DSIIDriver"="Installed".

8. Modify the driver definition for that driver. Change the line that says [HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\MyQuickstartDSIIDriver] so that it contains your new driver name. For example, [HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\<YourProjectName>DSIIDriver].
9. Click **Edit** > **Find and Replace** > **Quick Replace**. Replace "Quickstart" in the whole file with the name of your new ODBC driver.
10. Click **Save** and close the file.
11. In the Registry Editor (regedit.exe), click **File** > **Import**, navigate to the registry file that you just modified and click **Open**.

A message is displayed that says that the keys and values have been successfully added to the registry.

Viewing Your DSNs Using ODBC Data Source Administrator

To view the DSNs you created in the Windows Registry for your custom driver project:

1. Run the Windows ODBC Data Source Administrator. For 32-bit drivers on 32-bit Windows and 64-bit drivers on 64-bit Windows, open the Control Panel, select **Administrative Tools** > **Data Sources (ODBC)**. If your Control Panel is set to view by category, then Administrative Tools is located under System and Security.

   > **Note:** For 32-bit drivers on 64-bit Windows, you must use the 32-bit ODBC Data Source Administrator, located at C:\WINDOWS\SysWOW64\odbcad32.exe. See Appendix A: ODBC Data Source Administrator on Windows 32-Bit vs. 64-Bit for details.

2. In the ODBC Data Source Administrator, click the **System DSN** tab.
3. Scroll through the list of System Data Sources, select <YourProjectName>DSII and click **Configure**. The Data Source Configuration window opens and displays the data source name, description and the data directory.
4. Now that you have looked at the configuration information for your new driver, click **Cancel** to close the Data Source Configuration window.

Connecting to the Data Source Using Your Customizable Driver

To test your customizable driver using ODBC Test:

1. Start the ODBC Test tool. By default, the ODBC Test application is installed in the following folder: C:\Program Files (x86)\Microsoft Data Access SDK 2.8\Tools\. Navigate to the folder that corresponds to your driver's architecture (amd64, ia64 or x86) and click odbcte32.exe to launch the ANSI version or click odbct32w.exe to launch the Unicode version. It is important to run the correct version of the ODBC Test tool for ANSI or Unicode and 32-bit or 64-bit.
2. Attach Visual Studio to the ODBC Test process. To do this, go to Microsoft Visual Studio and then click **Debug** > **Attach to Process**.

3. In the Attach to Process window, select the ODBC Test process and click **Attach**. The process name will be either odbc32.exe or odbct32w.exe.
4. Add a breakpoint in Main_Windows.cpp, on the function *DSIDriverFactory()*. This function runs as soon as the Driver Manager loads the ODBC driver.
5. In the ODBC Test tool, select **Conn** > **Full Connect**. The Full Connect window opens.
6. Select your Data Source from the list of data sources and click **OK**. If you do not see your data source in the list, make sure that you are running the version of the ODBC Test tool that corresponds to the version of the data source that you created. In other words, if you created a 32-bit data source then you should be using the 32-bit version of the ODBC Test tool.
7. You should hit the breakpoint you created, and focus should switch to Visual Studio.
8. To continue running the program, select **Debug** > **Continue**. The focus returns to the ODBC Test window.

✓ **Day One Checkpoint – Verify Your Progress**

- Simba SDK is installed successfully
- QuickStart sample driver builds without errors
- Registry keys are visible in Registry Editor
- Data source appears in the ODBC Data Source Administrator
- Successfully connected to QuickstartDSII using the ODBC Test tool
- New project directory is created and files renamed
- New driver project builds without errors
- New driver's registry keys are imported successfully
- New driver connects successfully via the ODBC Test tool

Summary – Day One (Windows)

At this point, you have completed the following tasks:

- Installed Simba SDK and built the QuickStart sample driver included with Simba SDK.
- Learned how to create new driver definitions and Data Source Names in the Windows Registry.
- Viewed DSNs using the ODBC Data Source Administrator.
- Tested drivers using an ODBC-enabled application.

- Set up a new project for customization based on a sample driver project included in SimbaEngine.

Day One (Linux/Unix and macOS): Environment Setup

Today's task is to set up and test the development environment and project files for the connector. By the end of the day, you will have compiled and tested your first ODBC connector.

LEARNING OBJECTIVES

By the end of this section, you will be able to:

- Install and configure the Simba SDK on Linux, Unix, or macOS
- Build the QuickStart sample connector using makefiles
- Understand the ODBC driver manager configuration file structure
- Configure data sources in odbc.ini and connectors in odbcinst.ini
- Test ODBC connectivity using the iodbctest utility
- Set up a new project for your custom ODBC connector
- Debug your connector using GDB (Linux/Unix) or LLDB (macOS)
- Enable and read connector log files

The Day One instructions explain how to install the Simba SDK, compile the sample ODBC connector, and review the configuration information created at compile time.

After the sample ODBC connector is successfully compiled, it is used to retrieve data from the data source that is included with the Simba SDK. The sample ODBC connector is then used to create the framework for a custom ODBC connector, which is renamed and used to retrieve sample data.

At the end of the day, you will have compiled, built and tested your custom ODBC connector.

Install the Simba SDK

Linux/Unix

Simba SDK is provided as `SimbaEngineSDK[BUILD].tar.gz`, where [BUILD] is the build number and platform, for example:

```
SimbaEngineSDK_Release_Linux-x86_10.4.0.1000.tar.gz
```

This file is a .tar file that has been compressed using gzip.

The Simba SDK installation package includes:

- Simba SDK implementation in all supported platforms.
- Sample connectors for most supported platforms.

- PDF versions of the documentation.
- An HTML version of the C++ and Java API.

To install Simba SDK on Linux or Unix:

1. Uninstall any previous versions of the Simba SDK.
2. Ensure that the evaluation license is installed. See Appendix H: Install the Evaluation License.
3. Open a command prompt and navigate to the directory where the Simba SDK will be installed.
4. Copy the `SimbaEngineSDK[BUILD].tar.gz` file to the Simba SDK installation directory, where [BUILD] is the build number and platform. Ensure you copy the correct version of the product for your platform. To determine your machine version, type `uname -m` at the command prompt.
5. Uncompress the file by typing the following: `gunzip SimbaEngineSDK[BUILD].tar.gz`
6. Extract the .tar file by typing the following: `tar -xvf SimbaEngineSDK[BUILD].tar`
7. Verify the installation by listing the extracted content. `ls -la SimbaEngineSDK/10.4`

You should see the following directories:

Directory	Description
DataAccessComponents/	Core Simba SDK libraries and headers
Documentation/	Developer guides, API reference, and sample configuration files
Examples/	Sample driver source code, databases, and build files

Simba SDK is now installed.

macOS

Simba SDK for macOS platforms includes an installer to help you install and configure the SDK correctly. This installer removes any previous installations, installs the SDK, and configures the DSN and other configuration files for the sample connectors.

Important: If a previous installation of the Simba SDK with the same product and compiler version exists on your machine, the installer removes it. We recommend that you back up any previous work before reinstalling the SDK.

Note: If the ODBC configuration files `odbc.ini` and `odbcinst.ini` exist on your machine, the installer modifies them to add information for the sample connectors. Both the system-wide configuration files under /Library/ODBC and the user-specific files under ~/Library/ODBC are modified. ODBC configuration files in other locations are not modified. If these files do not exist on your machine, the installer creates them.

To install Simba SDK on macOS:

Simba SDK is distributed as a .zip file containing a pkg file or a a .tar.gz archive file.

1. Uninstall any previous versions of the Simba SDK.
2. Ensure the evaluation license is installed. See *Appendix H: Install the Evaluation License*.
3. Copy the `SimbaEngineSDK[BUILD].tar.gz` file to the SDK installation directory, where [BUILD] is the build number and platform. Ensure you copy the correct version of the SDK for your platform. To determine your machine version, type `uname -m` at the command prompt.
4. Uncompress the file by typing the following: `gunzip SimbaEngineSDK[BUILD].tar.gz`
5. Extract the .tar file by typing the following: `tar -xvf SimbaEngineSDK[BUILD].tar`
6. Double-click the `.pkg` file, then double-click the `.pkg` file. The installer launches.
7. In the Introduction window, select **Continue**.
8. In the License window, read the license agreement, then select **Continue**.
9. In the resulting window, if you agree with the license conditions, select **Agree**.
10. In the Installation Type window, if you agree with the install location, select **Install**. Otherwise, select **Change Install Location** and select a different disk.

    > **Note:** The installer always installs the Simba SDK to `/Library/Simba_XCode[XCode_VERSION]/SimbaEngineSDK`, but you can choose which disk. On macOS platforms, Microsoft Excel 2016 only loads connectors from `/Library` and `/Applications`.

11. Enter the username and password to complete the installation.In the final installation window, select **Close**.

The SDK is installed to the location `/Library/Simba_XCode[XCode_VERSION]/SimbaEngineSDK`.

The following system-wide environment variables are set using the plist file `/Library/LaunchAgents/setenv.SIMBAENGINESDK.plist`:

- SIMBAENGINE_DIR=`/Library/Simba_XCode[XCode_VERSION]/SimbaEngineSDK/10.4/DataAccessComponents`
- SIMBAENGINE_THIRDPARTY_DIR=`/Library/Simba_XCode[XCode_VERSION]/SimbaEngineSDK/10.4/DataAccessComponents/ThirdParty`

Build the Sample ODBC Connector

You can use the sample makefile to build the QuickStart connector. The sample makefile automatically detects the required settings based on your operating system, machine bitness, and compiler. For more information about different makefile options, see Compiling Your Connector in the guide Simba SDK Developer Guide NON SQL-Capable Data Sources.

Linux/Unix

The sample connectors included with the Simba SDK are installed in the folder `[INSTALL_DIR]/SimbaEngineSDK/10.4/Examples`, where [INSTALL_DIR] is the installation directory. The sample connectors include sample makefiles.

In the following instructions, replace [INSTALL_DIR] with the Simba SDK installation directory.

To build the SDK QuickStart sample connector on Linux or Unix:

1. Set the SIMBAENGINE_DIR environment variable: `export SIMBAENGINE_DIR=[INSTALL_DIR]/SimbaEngineSDK/10.4/DataAccessComponents`
2. Set the SIMBAENGINE_THIRDPARTY_DIR environment variable: export `SIMBAENGINE_THIRDPARTY_DIR=[INSTALL_DIR]/SimbaEngineSDK/10.4/DataAccessComponents/ThirdParty`
3. Change to the following directory: `[INSTALL_DIR]/SimbaEngineSDK/10.4/Examples/Source/Quickstart/Source`
4. Type ./mk.sh MODE=debug to run the makefile for the debug target.

> **Important:** Do not use the makefile directly. Use the mk.sh script instead. If you wish to use your own build system, see ${SIMBAENGINE_DIR}/Makefiles/GetBuildInfo.README.

The resulting library, libQuickStart<BITNESS>.so, is put in the following directory: `[INSTALL_DIR]/SimbaEngineSDK/10.4/Examples/Source/QuickStart/Bin/<BUILD>/<RELEASE|DEBUG><BITNESS>`.

Where <BUILD> is a combination of your operating system, machine bitness, and compiler, <RELEASE|DEBUG> is either release or debug, and <BITNESS> is 32 or 64.

Example Build Location

`[INSTALL_DIR]/SimbaEngineSDK/10.4/Examples/Source/QuickStart/Bin/Linux_x86_gcc/debug64/libQuickStart64.so`

macOS

The sample connectors included with the Simba SDK are installed in the folder `/Library/Simba_XCode[XCode_VERSION]/SimbaEngineSDK/10.4/Examples`. The sample connectors include sample makefiles.

In the following instructions, replace [INSTALL_DIR] with the Simba SDK installation directory, for example `/Library/Simba_XCode7`.

To build the SDK QuickStart sample connector on macOS:

1. Change to the following directory: `[INSTALL_DIR]/SimbaEngineSDK/10.4/Examples/Source/Quickstart/Source`
2. Type ./mk.sh MODE=debug to run the makefile for the debug target.

> **Important:** Do not use the makefile directly. Use the mk.sh script instead.

3. By default, the makefile detects the latest version of XCode on your machine. Optionally, you can specify a different version of XCode using the environment variable DEVELOPER_DIR. For an example, see Compiling Your Connector in Developing Connectors for Data Stores Without SQL.

The resulting library, libQuickStart<BITNESS>.dylib, is put in the following directory: `[INSTALL_`

DIR]/SimbaEngineSDK/10.4/Examples/Source/QuickStart/Bin/<BUILD>/<RELEASE|DEBUG><BITNESS>.

Where <BUILD> is a combination of your operating system, machine bitness, and compiler, <RELEASE|DEBUG> is either release or debug, and <BITNESS> is 64.

Example Build Location Using XCode

```
/Library/Simba_
XCode16/SimbaEngineSDK/10.4/Examples/Source/QuickStart/Bin/Darwin_
multiarch_Xcode16/debug64-universal/libQuickStart64.dylib
```

Configure the Connector and Data Source

You configure the Data Source Name (DSN) in the odbc.ini file and the connector in the odbcinst.ini file. The driver manager uses information in these files to locate and load ODBC connectors. You configure connector-specific information, such as log file location, in the `.simba.QuickStart.ini` file.

Driver Managers

On Linux and Unix platforms, the Simba SDK supports both the iODBC and the unixODBC driver managers. On macOS platforms, the SDK supports the iODBC and unixODBC driver manager.

Linux, Unix, and macOS platforms do not come with a driver manager as part of the operating system. You need to install your own driver manager before loading a connector and accessing data stores. For more information on installing a driver manager, see *Appendix C: Driver Managers*.

Linux/Unix

Using the Sample Configuration Files (Linux/Unix)

If the odbc.ini and odbcinst.ini files already exist on your machine, you can add the QuickStart connector entries to these files. If these files do not exist on your machine, you can create them and add the required entries. The Simba SDK includes sample files in the following directory: `[INSTALL_DIR]/SimbaEngineSDK/10.4/Documentation/Setup` where [INSTALL_DIR] is the Simba SDK installation directory.

Important:

- The sample odbc.ini and odbcinst.ini files show how to configure DSNs and connector entries for 32-bit connectors on Linux platforms. To configure a 64-bit connector, a debug connector, or a connector for platforms other than Linux, you will need to make the changes shown below.
- The sample .simba.quickstart.ini file is hidden, so you may need to configure your window or terminal listing to view hidden files.

Configuring an ODBC Data Source in the odbc.ini File (Linux/Unix)

ODBC Data Sources are defined in the .odbc.ini configuration file. If this file does not exist, you must create it in the correct location.

To create and configure the odbc.ini file:

1. To see if the .odbc.ini file already exists in the home directory, run the following command: `ls -al ~ | grep .odbc.ini`
2. If no response is returned, the .odbc.ini file does not exist in the home directory. To see if the file already exists in another location, check if the ODBCINI environment variable is set. If it is set, then check the specified location to see if the file exists.
3. If the file does not exist, copy the sample odbc.ini file to the desired directory. This example shows how to copy the file to the user's home directory, but you can choose another directory: `cp [INSTALL_DIR]/SimbaEngineSDK/10.4/Documentation/Setup/odbc.ini ~/.odbc.ini`
4. Using a text editor, open the `~/.odbc.ini` configuration file.

   Note: The "." before `odbc.ini` in `~/.odbc.ini` causes the copied file to be hidden. To open the file, you may need to configure your text editor to show hidden files.

5. Replace every instance of [INSTALLDIR] with the installation location of the Simba SDK.
6. Make sure there is an entry in the [ODBC Data Sources] section that defines the Data Source Name (DSN). The [ODBC Data Sources] section is used to specify the available data sources. Example:

   [ODBC Data Sources]

   QuickstartDSII=QuickstartDSIIDriver

7. Make sure there is a section with a name that matches the data source name (DSN). This section specifies the configuration options as key-value pairs. Be sure to use the path to the connector that you built in the previous step, as shown below. By default, the sample odbcinst.ini file contains a path to the pre-built connectors that ship with the Simba SDK.

Example: 32-bit Connector

```
[QuickstartDSII]
Description=Sample 32-bit Simba SDK QuickStart DSII
DBF=[INSTALL_DIR]/SimbaEngineSDK/10.4/Examples/Databases/Quickstart
Driver=[INSTALL_
DIR]/SimbaEngineSDK/10.4/Examples/Source/Quickstart/Bin/Linux_x86_
gcc/debug32/libQuickstart32.so
```

Example: 64-bit Connector

```
[QuickstartDSII]
Description=Sample 64-bit Simba SDK QuickStart DSII
DBF=[INSTALL_DIR]/SimbaEngineSDK/10.4/Examples/Databases/Quickstart/
Driver=[INSTALL_
DIR]/SimbaEngineSDK/10.4/Examples/Source/Quickstart/Bin/Linux_x86_
gcc/debug64/libQuickstart64.so
```

```
Locale=en-US
```

Note: Note: replace [INSTALL_DIR] with the installation directory.

Important: Take care not to include characters such as spaces or tabs at the end of a value. This can cause failure to connect.

Configuring an ODBC Connector in the odbcinst.ini File (Linux/Unix)

ODBC Connectors are defined in the odbcinst.ini configuration file. This configuration is optional, because connectors can be specified directly in the odbc.ini configuration file as discussed in the previous section. Before defining the connector, see if the odbcinst.ini file exists.

To configure a connector in the .odbcinst.ini file:

1. To see if the .odbcinst.ini file exists in your home directory, type the following command: `ls -al ~ | grep .odbcinst.ini`
2. If the file does not exist, then nothing is returned at the command prompt. To see if the file already exists in another location, check if the ODBCINSTINI environment variable is set. If it is set, then check the specified location to see if the file exists.
3. If the file does not exist, copy the sample odbcinst.ini file to the desired directory. This example shows how to copy the file to the user's home directory, but you can choose another directory: `cp [INSTALL_DIR]/SimbaEngineSDK/10.4/Documentation/Setup/odbcinst.ini ~/.odbcinst.ini`
4. Open the .odbcinst.ini configuration file in a text editor.
5. Replace every instance of [INSTALLDIR] with the installation location of the Simba SDK.
6. Add a new entry to the [ODBC Drivers] section. The [ODBC Drivers] section is used to specify the available connectors. Enter the connector name and the value Installed. The connector name should be used for the "Driver" value in the data source definition instead of the connector's shared library name. Example:

```
[ODBC Drivers]
QuickstartDSIIDriver=Installed
```

7. Add a new section with a name that matches the new connector's name. This section will contain the configuration options specified as key-value pairs. Be sure to use the path to the connector that you built in the previous step, as shown below.

Example - 32-bit Linux:

```
[QuickstartDSIIDriver]
```

```
APILevel=1

ConnectFunctions=YYY

Description=Sample 32-bit Simba SDK QuickStart DSII

Driver=[INSTALL_
DIR]/SimbaEngineSDK/10.4/Examples/Source/Quickstart/Bin/Linux_x86_
gcc/debug32/libQuickstart32.so

DriverODBCVer=03.80

SQLLevel=1
```

Note: Note: replace [INSTALL_DIR] with the installation directory.

Configuring the QuickStart Connector in the simba.ini File (Linux/Unix)

Configuration options for connectors created using the Simba SDK appear in the connector's configuration file, which by default is called `.simba.drivername.ini`. This file specifies connector-wide settings.

The Simba SDK searches for the connector `.ini` file in the following locations, in the specified order:

1. The path, including the file name, specified using the SIMBAINI environment variable.
2. The connector directory, as a non-hidden `.ini` file.
3. The directory that the client application is launched from.
4. In $HOME, as a hidden `.ini` file.
5. In /etc/ as a non-hidden `.ini` file.

Note: Examples in this section assume that the .ini file exists in the user's home directory.

To configure the QuickStart Connector in the .simba.quickstart.ini file:

1. To see if the .simba.quickstart.ini file already exists in your home directory, type the following command: `ls -al ~ | grep .simba.quickstart.ini`
2. If the file does not exist, copy the sample `.simba.quickstart.ini` file to the desired directory. Note that the sample `.simba.quickstart.ini` file is hidden. This example shows how to copy the file to the user's home directory, but you can choose another directory: `cp [INSTALL_DIR]/SimbaEngineSDK/10.4/Documentation/Setup/.simba.quickstart.ini ~/.simba.quickstart.ini`
3. Open the `~/.simba.quickstart.ini` configuration file in a text editor.
4. Replace every instance of [INSTALLDIR] with the installation location of the Simba SDK.
5. Edit the DriverManagerEncoding setting. If you are using the iODBC driver manager, set the DriverManagerEncoding setting to UTF-32. If you are using the unixODBC driver manager, check which setting to use: at a command prompt, type odbc_config --cflags. If you see the "DSQL_

WCHART_CONVERT" flag, then set DriverManagerEncoding to UTF-32; otherwise, set it to UTF-16. For more information about your ODBC driver manager, consult your system administrator or your ODBC Driver Manager documentation.

> **Note:**
>
> - This step is optional because the Simba SDK automatically detects the type and version of the driver manager.
> - If you are unsure about which driver manager you are using, or where the driver manager is installed, contact your system administrator or see Appendix C: Driver Managers.

6. Edit the ErrorMessagesPath setting to replace [INSTALLDIR] with your install directory.
7. Set the ODBCInstLib to the name/path of the ODBCInst shared library for the Driver Manager that you are using. The ODBCInst library is a part of the driver manager but is used by the connector to read values from the odbc.ini file. The value of this key is the name/path of the ODBCInst shared library and depends on which driver manager you are using. Note that this value will be passed directly to dlopen, so it may either be a filename (& resolved according to LD_LIBRARY_PATH/DYLD_LIBRARY_PATH/LIBPATH) or an absolute path.

   > **Note:** This step is optional, because the Simba SDK automatically detects the type and version of the ODBCInst library. Set this value only if you want to override the value that is automatically detected.

   Example: the iODBC Driver Manager (notice the i after lib)

   ```
   ODBCInstLib=<driver manager dir>/lib/libiodbcinst.so
   ```

   Example: the unixODBC Driver Manager

   ```
   ODBCInstLib=<driver manager dir>/lib/libodbcinst.so
   ```

   Save the file when you are done.

macOS

On macOS platforms, the installer configures the sample connectors for you. If the ODBC configuration files odbc.ini and odbcinst.ini exist on your machine, the installer modifies them to add information for the sample connectors. Both the system-wide configuration files under /Library/ODBC and the user-specific files under ~/Library/ODBC are modified. ODBC configuration files in other locations are not modified. If these files do not exist on your machine, the installer creates them.

To configure the QuickStart connector and data source on macOS:

1. In the /Library/ODBC directory, open the odbc.ini configuration file in a text editor.
2. Make sure there is an entry in the [ODBC Data Sources] section that defines the data source name (DSN) of the QuickStart sample connector. Example:

```
[ODBC Data Sources]

QuickstartDSII=QuickstartDSIIDriver
```

3. Make sure there is a section with a name that matches the data source name (DSN). Example (using XCode 7):

```
[QuickStartDSII]

Description=64-bit QuickStart DSII

DBF=/Library/Simba_
XCode7/SimbaEngineSDK/10.4/Examples/Databases/QuickStart

Driver=/Library/Simba_
XCode7/SimbaEngineSDK/10.4/Examples/Source/QuickStart/Bin/Darwin_x86_
Xcode7/debug64/libQuickStart64.dylib
```

4. Save and close the file.
5. Open the odbcinst.ini configuration file in a text editor.
6. Add a new entry to the [ODBC Drivers] section. Example:

```
[ODBC Drivers]

QuickStartDSIIDriver=Installed
```

7. Add a new section with a name that matches the new connector name. Example (using XCode 7):

```
[QuickStartDSIIDriver]

Driver=/Library/Simba_
XCode7/SimbaEngineSDK/10.4/Examples/Source/QuickStart/Bin/Darwin_x86_
Xcode7/debug64/libQuickStart64.dylib
```

8. Save and close the file.

The QuickStart connector and data source are configured.

Connect to the Data Source

For the QuickStart sample connector to connect to the sample database successfully, the library path environment variable must include references to the OpenSSL and ICU libraries.

OpenSSL is used by SimbaClient for ODBC. Your custom ODBC connector may not require this library.

Note:
You must have a driver manager installed. See Appendix C: Driver Managers.

You must use a 64-bit driver manager with a 64-bit connector, or a 32-bit driver manager with a 32-bit connector.

Linux/Unix

To add the ICU and OpenSSL libraries on Linux or Unix, type the following at the command line, replacing [INSTALLDIR], <PLATFORM>, and <CONFIGURATION>. Note these libraries are supported for all versions of Unix and Linux, not just for CentOS:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:
[INSTALLDIR]/SimbaEngineSDK/10.4/DataAccessComponents/ThirdParty/icu/78.1/<PLATFORM>/<CONFIGURATION>/lib:
[INSTALLDIR]/SimbaEngineSDK/10.4/DataAccessComponents/ThirdParty/openssl/3.5/<PLATFORM>/<CONFIGURATION>/lib
```

This document uses the LD_LIBRARY_PATH environment variable for Linux and Unix platforms.

macOS

On macOS, the DYLD_LIBRARY_PATH environment variable must include references to the OpenSSL and ICU libraries.

To add the OpenSSL library and the ICU library to the library path:

1. Locate the path to the correct version of the OpenSSL library for your platform, compiler, and machine bitness. For example:

   ```
   /Library/Simba_XCode16/SimbaEngineSDK/10.4/DataAccessComponents/ThirdParty/openssl/3.5/Darwin_multiarch_Xcode16/release64-universal/lib
   ```

2. Add the OpenSSL path to the DYLD_LIBRARY_PATH environment variable. For example:

   ```
   export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:[PathtoOpenSSL]
   ```

3. Locate the path to the correct version of the ICU library for your platform, compiler, and machine bitness. For example:

   ```
   /Library/Simba_XCode16/SimbaEngineSDK/10.4/DataAccessComponents/ThirdParty/icu/78.1/Darwin_multiarch_Xcode16/release64-universal/lib
   ```

4. Add the ICU path to the DYLD_LIBRARY_PATH environment variable. For example:

   ```
   export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:[PathtoICU]
   ```

Linux/Unix and macOS

To test connecting the QuickStart sample connector to the sample data source:

This procedure uses the iodbctest utility that is included with the iODBC driver manager. For help, see Appendix I: Troubleshooting.

1. At the command prompt, type iodbctest.
2. At the prompt that says “Enter ODBC connect string”, type ? to show the list of DSNs and Drivers. The list contains the QuickStartDSII DSN.
3. To connect to your data source, type: DSN=QuickStartDSII. The prompt SQL> appears.
4. Type a SQL command to query the database. For example: `SELECT * FROM PRODUCT`
5. The SQL results are returned. To quit iodbctest, type quit at the prompt.

Note: Note that Microsoft Excel 2016 will only load connectors from /Library and /Applications.

Once the QuickStart sample connector project is built, it can be used to create a custom connector project.

Set up a Custom ODBC Connector Project

Once the QuickStart project has been built and tested, you can create a custom project for your ODBC connector. The procedure below applies to Linux, Unix, and macOS platforms.

Important: It is very important that you create your own project directory. You might be tempted to simply modify the sample project files, but we strongly recommend that you create your own project directory. If you simply modify the sample project files:

- All your changes will be lost when you install a new version of the SDK.
- You will lose your frame of reference for debugging. There may be times, for debugging purposes, that you will need to see if the same error occurs using the sample connectors. If you have modified the sample connectors, this won’t be possible.

To create a custom ODBC connector project based on the QuickStart sample connector:

1. Ensure you are working in a window or shell where the SIMBAENGINE_DIR and SIMBAENGINE_THIRDPARTY_DIR environment variables are set, as explained in Build the Sample ODBC Connector. (On macOS, these are set system-wide by the installer.)
2. Copy the Quickstart directory to create a new top-level directory for your custom ODBC connector project. Be sure to copy hidden files and symlinks too, for example:

```
mkdir [INSTALL_DIR]/SimbaEngineSDK/10.4/Examples/Source/MyQuickstart

cp -a [INSTALL_DIR]/SimbaEngineSDK/10.4/Examples/Source/Quickstart/.
[INSTALL_DIR]/SimbaEngineSDK/10.4/Examples/Source/MyQuickstart
```

Note: Replace [INSTALL_DIR] with the Simba SDK installation directory (on macOS, for example, /Library/Simba_XCode7). This new directory is referred to as the [Project] directory in the following steps.

3. In the [Project]/Source directory, open the GNUmakefile in a text editor.

4. Replace the target.driver target name with the name of your custom connector. Example:

```
Replace this line: target.driver = libQuickstart${BITS}.${SO}
With this line:    target.driver = libMyQuickstart${BITS}.${SO}
```

5. Save and close the file.
6. In the [Project]/Source directory, run the following command to build your custom ODBC connector: `./mk.sh MODE=debug`

Your custom ODBC connector project is built.

Configure Your Custom Connector and Data Source

Linux/Unix

Simba SDK supports both the iODBC and the unixODBC driver managers. The driver manager uses configuration files to define and configure ODBC data sources and connectors. The odbc.ini file defines ODBC data sources, or DSNs, and the odbcinst.ini file defines ODBC connectors. The `.simba.quickstart.ini` file defines connector-specific properties such as logging.

Note: The examples in this section assume your custom ODBC connector is a 32-bit connector named MyQuickStart.

To configure the custom connector and data source on Linux or Unix:

1. In the user's home directory, open the .odbc.ini configuration file in a text editor.
2. Make sure there is an entry in the [ODBC Data Sources] section that defines the data source name (DSN) of your custom ODBC connector. Example:

```
[ODBC Data Sources]
MyQuickstartDSII=MyQuickstartDSIIDriver
```

3. Make sure there is a section with a name that matches the data source name (DSN). Example (32-bit Linux):

```
[MyQuickStartDSII]
Description=32-bit MyQuickStart DSII
DBF=[INSTALL_DIR]/SimbaEngineSDK/10.4/Examples/Databases/QuickStart
Driver=[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Builds/Bin/Linux_x86_
gcc/release32/libMyQuickStart32.so
```

4. Open the .odbcinst.ini configuration file in a text editor.

5. Add a new entry to the [ODBC Drivers] section. Example:

```
[ODBC Drivers]
MyQuickStartDSIIDriver=Installed
```

6. Add a new section with a name that matches the new connector name. Example (32-bit Linux):

```
[MyQuickStartDSIIDriver]
Driver=[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Builds/Bin/Linux_x86_
gcc/release32/libMyQuickStart32.so
```

Your custom ODBC connector and data source are configured.

macOS

On macOS platforms, the Product supports the iODBC driver manager. This driver manager uses configuration files to define and configure ODBC data sources and connectors. The odbc.ini file is used to define ODBC data sources and the odbcinst.ini file is used to define ODBC connectors. Connector-specific information, such as log file location, is configured in the .simba.QuickStart.ini file.

To configure the odbc.ini file:

1. Open the /Library/ODBC/odbc.ini configuration file in a text editor.
2. Make sure there is an entry in the [ODBC Data Sources] section that defines the data source name (DSN). Example:

```
[ODBC Data Sources]
MyQuickStartDSII=MyQuickStartDSIIDriver
```

3. Make sure there is a section with a name that matches the data source name (DSN). Example:

```
[MyQuickStartDSII]
Description=Sample Simba SDK QuickStart DSII
DBF=[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Databases/QuickStart/
Driver=/Library/Simba_
XCode7/SimbaEngineSDK/10.4/Examples/Source/MyQuickStart/Bin/Darwin_x86_
Xcode7/debug64/libMyQuickStart64.dylib
Locale=en-US
```

4. Save and close the file.

To configure the odbcinst.ini file:

1. Open the /Library/ODBC/odbcinst.ini configuration file in a text editor.
2. Add a new entry to the [ODBC Drivers] section. For example:

```
[ODBC Drivers]
MyQuickStartDSIIDriver=Installed
```

3. Add a new section with a name that matches the new connector name. Example:

```
[MyQuickStartDSIIDriver]
Driver=/Library/Simba_
XCode7/SimbaEngineSDK/10.4/Examples/Source/MyQuickStart/Bin/Darwin_x86_
Xcode7/debug64/libMyQuickStart64.dylib
```

4. Save and close the file.

To configure the .simba.QuickStart.ini file:

1. Copy the sample `.simba.QuickStart.ini` file to the user's home directory. Note that the sample `.simba.QuickStart.ini` file is hidden.
2. Open the `~/.simba.QuickStart.ini` configuration file in a text editor.
3. Replace every instance of [INSTALLDIR] with the installation location of the Product.
4. Set the DriverManagerEncoding setting to UTF-32.

> **Note:**
> - This step is optional, because the Product automatically detects the type and version of the driver manager. Set the DriverManagerEncoding only if you want to override the value that is automatically detected.
> - If you are unsure where the driver manager is installed, contact your system administrator or see Appendix C: Driver Managers.

5. Edit the ErrorMessagesPath setting to replace [INSTALLDIR] with your install directory.
6. Set the ODBCInstLib to the absolute path of the ODBCInst library for the Driver Manager that you are using. For the iODBC Driver Manager this would be <driver manager dir>/lib/libiodbcinst.dylib (notice the 'i' after the lib).

> **Note:** This step is optional, because the Product automatically detects the type and version of the ODBCInst library. Set this value only if you want to override the value that is automatically detected.

7. Save the file.

The custom connector and data source are configured.

Debug Your Custom Connector

Stepping through your connector in a debugger shows how the Simba ODBC layer calls into your DSI implementation. The driver manager loads your connector only when an application makes a connection, not when the application starts. You can set breakpoints before or after that point. If you set them before, they may show as unresolved until the connector loads.

Test Application and Connection

Use **iodbctest**, which comes with the iODBC driver manager, as the host application. Use the following connection string on all platforms:

```
DSN=<YourProjectName>DSII;UID=<YourUserName>;PWD=<YourPassword>
```

- Before debugging, check that iodbctest can connect to your custom DSN. Follow Connect to the Data Source, using your DSN in place of QuickStartDSII. Type quit to exit.
- You can start the connection in two ways: type the connection string at the iodbctest Enter ODBC connect string prompt, or pass it as the iodbctest argument when you launch it under the debugger.
- Omit UID and PWD if your connector does not require credentials.

Recommended Breakpoints

Entry point	File	When it is triggered	TODO
Simba::DSI::DSIDriverFactory()	Main_Unix.cpp	As soon as the driver manager loads the connector library on the first connection. Set this breakpoint first: hitting it confirms that the connector is loading.	#1
QSDriver::SetDriverPropertyValues()	QSDriver.cpp	When the connector singleton is built and its general properties are set.	#2
QSConnection::UpdateConnectionSettings ()	QSConnection.cpp	After the ODBC layer parses the connection string and merges it with the DSN entries, to check the settings.	#5
QSConnection::Connect()	QSConnection.cpp	After UpdateConnectionSettings () reports no missing required settings.	#6
QSDataEngine::MakeNewMetadataTable()	QSDataEngine.cpp	When the application calls an ODBC catalog function.	#7
QSDataEngine::OpenTable()	QSDataEngine.cpp	While a SQL statement is being prepared, once for each table the SQL Engine	#8

Entry point	File	When it is triggered	TODO
		opens for the query.	

A connection with iodbctest hits the first four entry points in order. At the SQL> prompt, a query such as `SELECT * FROM <table> triggers OpenTable()`.

This step confirms that your custom connector, based on the QuickStart project, is installed and configured correctly and that your development system is set up properly.

Enable Logging

You can turn on logging for your custom ODBC connector. By setting the log level to Trace, you can gain a better understanding of how your custom ODBC connector works. This procedure applies to Linux, Unix, and macOS platforms.

To enable logging in your custom ODBC connector:

1. Open the `.simba.QuickStart.ini` file in the user's home directory.

   Note: Your custom ODBC connector is not yet completely rebranded, so configuration information is read from the `.simba.QuickStart.ini` file. The name of this file is set using #define DRIVER_LINUX_BRANDING in the Main_Unix.cpp file.

2. Set the LogLevel to 6 for trace level, or another level if you prefer.
3. Set the LogPath to the directory to use for log files. Example:

```
LogLevel=6
LogPath=/usr/tmp/myquickstart/logs
```

The log files are created the next time the connector is used.

Log File Format

The log files have the following format, where [Message] is optional:

```
[Date] [Log Level] [Thread ID] [Class] [Message]
```

Example:

```
Jun 15 14:05:12.017 INFO 9864 ConnectionSettings::LoadSettings: ConnString
setting: "DSN" = "MyQuickstartDSII"
```

TODO Items Reference

When you build your custom connector project, "TODO" messages appear in the build output. Over the next four days, you will visit each TODO and modify the source code. The following table maps each TODO item to the day of this guide in which it is covered:

TODO #	Description and Day
1	Construct connector singleton – Day 2
2	Set the connector properties – Day 2
3	Set the connector-wide logging details – Day 2
4	Set the connection-wide logging details – Day 2
5	Check Connection Settings – Day 2
6	Establish A Connection – Day 2
7	Create and return your Metadata Sources – Day 3
8	Open A Table – Day 4
9	Register Messages xml file for handling by DSIMessageSource – Day 5
10	Set the vendor name, which will be prepended to error messages – Day 5

✓ **Day One Checkpoint – Verify Your Progress**

- Simba SDK is installed successfully
- Environment variables (SIMBAENGINE_DIR, SIMBAENGINE_THIRDPARTY_DIR) are set correctly
- QuickStart sample connector builds without errors
- Configuration files (odbc.ini, odbcinst.ini) are properly configured
- Connector settings file (.simba.quickstart.ini) is configured
- Successfully connected to QuickStartDSII using iodbctest
- Can execute a SELECT query and see results
- New project directory is created and the GNUmakefile target renamed
- New connector project builds without errors
- New connector's configuration files are updated
- New connector connects successfully
- Debugger breakpoint in DSIDriverFactory() is hit (gdb or lldb)
- Logging is enabled and log files are created

Summary – Day One (Linux/Unix and macOS)

At this point, you have completed the following tasks:

- Installed the Simba SDK on Linux, Unix, or macOS and built the QuickStart sample connector. This verifies that your installation and development environment are properly configured.

- Learned about the ODBC driver manager configuration files (odbc.ini, odbcinst.ini) and how Data Source Names and connectors are configured.
- Tested the sample connector using the iodbctest utility.
- Created, built, and tested a custom connector project by copying the QuickStart connector. You can use this project as a framework to create your custom ODBC connector.
- Debugged your connector using GDB (Linux/Unix) or LLDB (macOS), and enabled logging.

Day Two: Connection Management

Today's goal is to customize the connector, enable logging, and establish a connection to your data store. To accomplish this you will visit TODO items 1 to 6. The instructions in this chapter apply to all platforms; platform-specific steps are identified with a platform heading.

LEARNING OBJECTIVES

By the end of this section, you will be able to:

- Understand and modify the DSIDriverFactory() entry point
- Set the configuration branding for your connector
- Configure connector properties and character encodings
- Set up connector-wide and connection-wide logging
- Validate and process connection settings
- Establish authenticated connections to your data store

Note: On Windows, when you build the project you will see the TODO messages in the Output window. To rebuild the whole solution, select Build > Rebuild Solution. If the Output window is not open, then select Debug > Windows > Output. Double-click a TODO message to jump to the relevant section of code. On Linux, Unix, and macOS platforms, open the source file named in each procedure and navigate to the TODO comment.

Construct the Connector Singleton (Set the Configuration Branding)

TODO #1: Construct connector singleton.

The `DSIDriverFactory()` implementation is the main entry point that is called from Simba's ODBC layer to create an instance of the DSI implementation. This method is called as soon as the Driver Manager calls `LoadLibrary()` on the ODBC connector.

Windows

On Windows platforms, `DSIDriverFactory()` is implemented in `Main_Windows.cpp`.

To construct the connector singleton:

1. Launch Microsoft Visual Studio 2022.
2. Click **File** > **Open** > **Project/Solution**.
3. Navigate to `[INSTALLDIR]\SimbaEngineSDK\10.4\Examples\Source\<YourProjectName>\Source and then open the <YourProjectName>.vcxproj` file.

4. Rebuild your solution and then double-click the TODO #1 message to jump to the relevant section of code. The Main_Windows.cpp file opens.
5. Look at the `DSIDriverFactory()` implementation.
6. Specify the location that is used when reading driver settings from the registry. This change is related to rebranding. Replace “Simba” with your company name and change “Quickstart” to the name of your driver in the following line:
```
SimbaSettingReader::SetConfigurationBranding("Simba\\Quickstart");
```
7. This step, like those in Day One, is important to distinguish your driver from our sample and other drivers. Add the processing, if you are building a commercial connector.
8. Click **Save**.

Linux/Unix and macOS

On Linux, Unix, and macOS platforms, `DSIDriverFactory()` is implemented in `Main_Unix.cpp`.

To construct the connector singleton:

1. In your custom ODBC connector project, open the file Main_Unix.cpp.
2. Navigate to the line **TODO #1: Construct connector singleton**.
3. Look at the `DSIDriverFactory()` implementation, and locate the following line of code:
```
SimbaSettingReader::SetConfigurationBranding(DRIVER_LINUX_BRANDING);
```
4. In the default implementation, DRIVER_LINUX_BRANDING defines the string "`simba.QuickStart.ini`". This is the name of the .ini file that specifies the connector settings.
5. Change this string to the name of a configuration file reflecting the name of your connector or company.
6. Save the `Main_Unix.cpp` file.
7. Update the name of the `simba.QuickStart.ini` file to match DRIVER_LINUX_BRANDING.

Set Connector Properties

TODO #2: Set the connector properties.

To set connector properties:

1. Open the file `QSDriver.cpp` and navigate to the line **TODO #2 Set the connector properties**. (On Windows, you can double-click the TODO #2 message in the Output window to jump to the relevant section of code.)
2. Go to the method `SetDriverPropertyValues()` , where the general properties for the connector are set. Change the properties described below:

Property	Description
DSI_DRIVER_ DRIVER_NAME	Set this property to the name of the connector (the same name used to replace QuickStartDSII in Day One). This is the connector name that is shown to the application.
DSI_DRIVER_	Optional. The encoding of char data from the perspective of the data store.

Property	Description
STRING_DATA_ ENCODING	Depending on the character sets, this property may need to be changed. The default value is ENC_UTF8.
DSI_DRIVER_ WIDE_STRING_ DATA_ENCODING	Optional. The encoding of wide character data from the perspective of the data store. Depending on the character sets, this property may need to be changed. The default is ENC_UTF16_LE.

Set Logging Details

TODO #3: Set the connector-wide logging details.

TODO #4: Set the connection-wide logging details.

This section explains how to set the connector-wide and connection-wide logging. Here you will customize how the connector logs errors and other information. The important loggers are the connector log, for anything not specific to a single connection, and the connection log, for anything unique to a single connection or statement within a connection. You can use the provided logger implementation and just rename the output file name, or you may entirely replace it later with your own implementation of the ILogger interface.

To set logging details:

1. Open the file `QSDriver.cpp` and navigate to the line **TODO #3 Set the connector-wide logging details**.
2. Change the connector log's file name.
3. Open the file `QSConnection.cpp` and navigate to the line **TODO #4 Set the connection-wide logging details**.
4. Change the connection log's file name. Under default behavior this is routed to the connector logs. If you want to log connections differently you need to override the `QSConnection::GetLog()` function.
5. Click **Save All**.

> **Note:** By default, the QuickStart connector maintains two kinds of log files: one for all connector-based calls and one for each connection created. Update these sections if you do not require such fine granularity in logging.

For more information about how to enable logging, see *Developing Connectors for Data Stores Without SQL*.

Check Connection Settings

TODO #5: Check Connection Settings.

When the Simba ODBC layer is given a connection string from an ODBC-enabled application, the Simba ODBC layer parses the connection string into key-value pairs. The entries in the connection string and the DSN are then sent to the `QSConnection::UpdateConnectionSettings()` function for validation.

If entries of the connection string overlap entries from the DSN, then the connection string will override parameters from the DSN. To pass additional parameters to your DSII, simply add new parameters to the connection string, or add new entries to the DSN entry. These values will automatically be picked up by the SDK and passed through for use by your DSII.

`UpdateConnectionSettings()` receives all the incoming connection settings that are specified in the DSN that was used to establish the connection. The role of this function is to ensure that all of the required, and any optional, settings are present. Note that actual data validation of the settings should be done in the `Connect()` function.

Example:

The connection string `DSN=QuickStart;UID=user;` will be broken down into key value pairs and passed in via the DSIConnSettingRequestMap parameter. In this case that map would contain two entries: `{DSN, QuickStart}` and `{UID, user}`. If a DSN was specified, then the DSN value is removed from the map and any entries that are stored in the preconfigured DSN are inserted into the map. Once the map has been created with all the key-value pairs from the connection string and DSN, this map is passed down to the DSII.

To check the connection settings for the custom connector:

1. Open the file `QSConnection.cpp` and navigate to the line **TODO #5 Check Connection Settings**.
2. Modify the `UpdateConnectionSettings()` function to validate that the settings (key-value pairs) in the DSIConnSettingRequestMap are sufficient to create a connection. Any settings that are not present should be added to the DSIConnSettingResponseMap parameter. We recommend using the `VerifyRequiredSetting()` or `VerifyOptionalSetting()` functions to perform this verification. These functions also add missing settings to DSIConnSettingResponseMap.
3. If any required values are missing, the connector will either fail to connect, or will call `PromptDialog()`, depending on the connection settings. If all required values exist, then `Connect()` will be called.
4. If any of the values received are invalid, then the code should throw an ErrorException seeded with DIAG_INVALID_AUTH_SPEC.

Note: The connection settings listed in `UpdateConnectionSettings()` are specific to the QuickStart connector. A custom connector will require different settings.

Example - QuickStart connector:

The QuickStart connector verifies that the settings contained in in_connectionSettings are sufficient to create a connection by using the following code:

```
VerifyRequiredSetting(QS_DBF_KEY, in_connectionSettings, out_
connectionSettings);

VerifyOptionalSetting(QS_DEFAULT_MAX_COLUMN_SIZE_KEY, in_connectionSettings,
out_connectionSettings);

VerifyOptionalSetting(QS_USE_CUSTOM_STATES_KEY, in_connectionSettings, out_
connectionSettings);

VerifyOptionalSetting(QS_USE_CACHING_KEY, in_connectionSettings, out_
connectionSettings);
```

```
VerifyOptionalSetting(QS_LOCALE, in_connectionSettings, out_
connectionSettings);
```

The QuickStart connector requires a single key in the DSN, DBF, which represents the file location to be searched. The other keys are optional. This QS_DBF_KEY is the connection key to use when looking up the DBF path in the connection string.

Manually Verifying the Connection Settings

Settings can also be verified manually. If the entries within in_connectionSettings are not sufficient to create a connection, then the connector can ask for additional information from the ODBC-enabled application by manually specifying the additional, required settings in out_connectionSettings. If there are no further entries required, simply leave out_connectionSettings empty.

Establish a Connection

TODO #6: Establish A Connection.

The Product calls `UpdateConnectionSettings()` before calling `QSConnection::Connect()`. Once `QSConnection::UpdateConnectionSettings()` returns out_connectionSettings without any required settings, if there are only optional settings, a connection can still occur, then the Simba ODBC layer calls `QSConnection::Connect()`, passing in all the connection settings received from the application.

During `Connect()`, the connector should have all the settings necessary to make a connection as verified by `UpdateConnectionSettings()`. You can use the utility functions `GetRequiredSetting()` and `GetOptionalSetting()` to request the required and optional settings for your connection, and attempt to make an actual connection.

To establish a connection:

1. Open the file `QSConnection.cpp` and navigate to the line **TODO #6 Establish A Connection.**
2. Look at the code that authenticates the user against your data store using the information provided within the in_connectionSettings parameter. Use `GetRequiredSetting()` and `GetOptionalSetting()` to access the settings in the map. Use the obtained values (e.g. hostname, username, password, etc.) to make a connection with your data source by passing them to your relevant API or network protocol.
3. Add validation to your custom ODBC connector. If authentication fails, throw an ErrorException seeded with DIAG_INVALID_AUTH_SPEC. Note that the sample ODBC connector does not perform validation.

The user is now authenticated against your data store.

✓ **Day Two Checkpoint – Verify Your Progress**

- Connector branding is updated with your company/connector name (TODO #1)
- DSI_DRIVER_DRIVER_NAME is set to your connector name (TODO #2)

- Character encoding settings are configured appropriately
- Log file names are customized for your connector (TODO #3 and #4)
- Connection settings validation is implemented in UpdateConnectionSettings() (TODO #5)
- Connect() authentication logic is implemented (TODO #6)
- Connector builds without errors
- Connector connects successfully with proper credentials

Summary – Day Two

At this point, you have completed the following tasks:

- Customized the configuration branding so your connector reads its own settings instead of the QuickStart sample settings.
- Set the connector properties, including the connector name and character encodings.
- Customized connector-wide and connection-wide logging.
- Implemented validation of the connection settings received from the connection string and DSN.
- Established an authenticated connection to your data store.

Day Three: Metadata Implementation

Today's goal is to return the data used to pass catalog information back to the ODBC-enabled application. The instructions in this chapter apply to all platforms.

LEARNING OBJECTIVES

By the end of this section, you will be able to:

- Understand ODBC catalog functions and their purpose
- Implement metadata sources for SQLGetTypeInfo
- Return catalog, schema, and table type information using the metadata helper
- Implement table and column metadata
- Map your data store types to standard ODBC SQL types

Create and Return Metadata Sources

Your custom ODBC connector uses metadata sources, provided by the Product, to handle SQL catalog functions.

Overview of SQL Catalog Functions

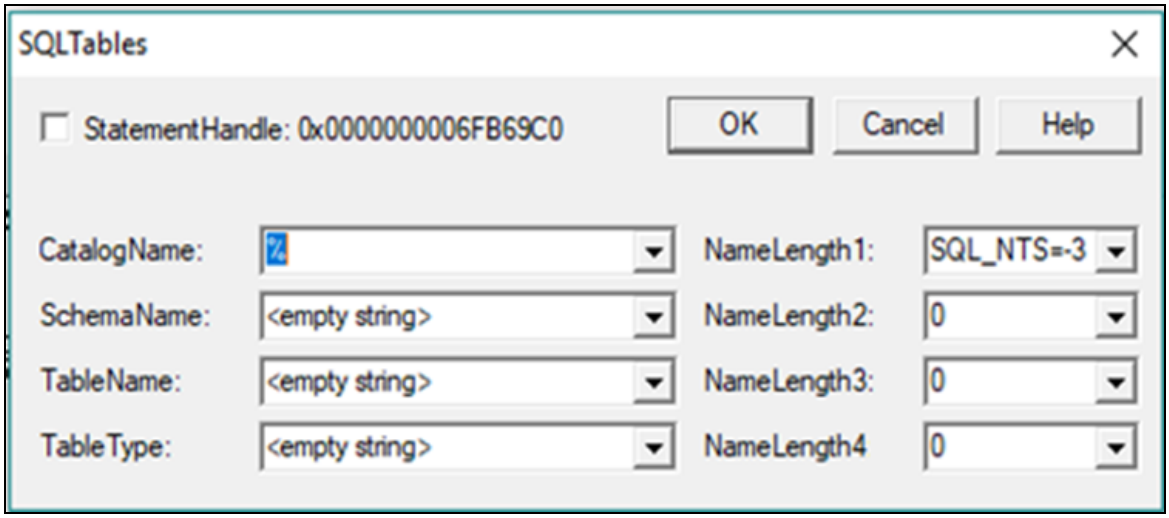
ODBC applications need to understand the structure of a data store in order to execute SQL queries against it. This information is provided using catalog functions. For example, an application might request a result set containing information about all the tables in the data store, or all the columns in a particular table. Each catalog function returns data as a result set.

Most ODBC-enabled applications require a connector to implement the following catalog functions. You may wish to implement additional catalog functions in your custom connector.

Catalog Function	Description
SQLGetTypeInfo	Returns information about data types supported by the data source.
SQLTables (CATALOG_ ONLY)	If CatalogName is SQL_ALL_CATALOGS and SchemaName and TableName are empty strings, the result set contains a list of valid catalogs for the data source. (All columns except the TABLE_CAT column contain NULLs.)
SQLTables (SCHEMA_ ONLY)	If SchemaName is SQL_ALL_SCHEMAS and CatalogName and TableName are empty strings, the result set contains a list of valid schemas for the data source. (All columns except the TABLE_SCHEM column contain NULLs.)
SQLTables (TABLE_TYPE_ ONLY)	If TableType is SQL_ALL_TABLE_TYPES and CatalogName, SchemaName, and TableName are empty strings, the result set contains a list of valid table types for the data source. (All columns except the TABLE_TYPE column contain NULLs.)
SQLTables	Returns the list of table, catalog, or schema names, and table types, stored in a specific data source.
SQLColumns	Returns a list of columns in one or more tables.

Example: Using Catalog Functions with the QuickStart Connector

1. In the ODBC Test application, connect to the QuickStart connector.
2. To send the SQLTables (CATALOG_ONLY) catalog function, select **Catalog** > **SQLTables**.
3. Enter ('%') for the CatalogName, then select the correct value for NameLength1. For example:



SQLTables catalog function in the ODBC Test application

4. Click **OK**.
5. Select to retrieve the results. The following list of valid columns for the QuickStart data source is returned: "TABLE_CAT", "TABLE_SCHEM", "TABLE_NAME", "TABLE_TYPE", "REMARKS"

For more information on SQL catalog functions, see https://msdn.microsoft.com/en-us/library/ms713520(v=vs.85).aspx.

Implementing Metadata Sources to Handle Catalog Functions

TODO #7: Create and return your Metadata Sources.

SQL catalog functions are represented in the DSI by metadata sources: there is one metadata source for each of the catalog functions.

`QSDataEngine::MakeNewMetadataTable()` is responsible for creating metadata sources. Metadata sources are used to return the catalog metadata about your data store to the ODBC application for the ODBC catalog functions.

Open the file `QSDataEngine.cpp` and navigate to the line **TODO #7 Create and return your Metadata Sources.** (On Windows, you can double-click the TODO #7 message in the Output window.)

There is one metadata source for each of the catalog functions. For example, when the application calls `SQLColumns()`, a DSI_COLUMNS_METADATA source is created to return the list of columns in one or more tables in the data store.

Each ODBC catalog function is mapped to a unique DSIMetadataTableId, which is then mapped to an underlying MetadataSource that the connector implements and returns. Each MetadataSource instance is responsible for the following:

1. Creating a data structure that holds the data relevant for the custom data store: Constructor.
2. Navigating the structure on a row-by-row basis: Move().
3. Retrieving data: GetData() (see Appendix E: Data Retrieval for a brief overview of data retrieval).

Required Metadata Sources

All custom ODBC connectors must implement the following metadata sources, as they are required by ODBC applications:

Metadata Source	Description
DSI_TABLES_ METADATA	List of all tables defined in the data source. This source is constructed via the QSMetadataHelper and SQL Engine.
DSI_ CATALOGONLY_ METADATA	List of all catalogs defined in the data source, if catalogs are supported. This source is constructed via the QSMetadataHelper and SQL Engine.
DSI_SCHEMA_ METADATA	List of all schemas defined in the data source. This source is constructed via the QSMetadataHelper and SQL Engine. QuickStartDSII does not support schemas.
DSI_ TABLETYPEONLY_ METADATA	List of all table types (TABLE, VIEW, SYSTEM) defined within the data source. This source is constructed via the QSMetadataHelper and SQL Engine.
DSI_COLUMNS_ METADATA	List of all columns defined across all tables in the data source. This source is constructed via the QSMetadataHelper and SQL Engine.
DSI_TYPE_INFO_ METADATA	List of the supported types by the data source. This means the actual types that can be stored in the data source, not necessarily the types that can be returned by the connector. For instance, a conversion may result in a type being returned that is not stored in the data source.

Most catalog types are created using the metadata helper in conjunction with the SQL Engine.

Handling DSI_TYPE_INFO_METADATA

SQLGetTypeInfo is used by applications to discover data types supported by your connector. The SDK supports all the types listed below, but you may want to modify this metadata source if your tables don't support storing all of them or if some of the default metadata differs from the defaults.

The underlying ODBC catalog function SQLGetTypeInfo is handled as follows:

1. When called with DSI_TYPE_INFO_METADATA, `QSDataEngine::MakeNewMetadataTable()` will return an instance of `QSTypeInfoMetadataSource()`.
2. The QuickStart sample connector exposes support for all data types, but due to its underlying file format, it is constrained to support only the following types:

SQL_BIGINT	SQL_BIT	SQL_CHAR

SQL_DECIMAL	SQL_DOUBLE	SQL_INTEGER
SQL_LONGVARCHAR	SQL_LONGWVARCHAR	SQL_NUMERIC
SQL_REAL	SQL_SMALLINT	SQL_TINYINT
SQL_TYPE_DATE	SQL_TYPE_TIME	SQL_TYPE_TIMESTAMP
SQL_VARCHAR	SQL_WCHAR	SQL_WVARCHAR

3. For your connector, you may need to change the types returned and the parameters for the types in `QSTypeInfoMetadataSource::PrepareType()`. Change the passed in SqlTypeInfo object to modify the parameters of the types that are supported.

Handling the Other MetadataSources

The other ODBC catalog functions, including SQLTables (CATALOG_ONLY), SQLTables (TABLE_TYPE_ONLY), SQLTables (SCHEMA_ONLY), SQLTables and SQLColumns, are used by applications to learn about your table structure (that is, their names and contents). They are handled automatically by the metadata helper class.

When these functions are called with any other DSIMetadataTableId, `QSDataEngine::MakeNewMetadataTable()` should return null. Returning null tells the Product to use the metadata helper class returned via `QSDataEngine::CreateMetadataHelper()` along with some default MetadataSources to create the data source metadata. You can also choose to return a DSIMetadataSource if you don't want to use the metadata helper.

Change the following methods:

- `QSMetadataHelper::QSMetadataHelper()`: The example constructor retrieves a list of the tables in the data source. Modify this method to load the tables defined in your data store.
- `QSMetadataHelper::GetNextTable()`: In the QuickStart connector, this method returns the next table in the data source. Modify this method to retrieve the next table from your data store.

The DSIExtMetadataHelper class works by retrieving the identifying information for each table and then opening the table via `QSDataEngine::OpenTable()`. After QSTable is implemented, the correct metadata will be returned for all the tables and columns in your data source.

You can now retrieve type metadata from your data store.

Tip: On Linux, Unix, and macOS platforms, this metadata is available using the datatypes command in the iodbctest utility. As well, the SQLTables catalog function is available using the tables command. You can use these commands to test your implementation of Day Three.

See also Fetching Metadata for Catalog Functions in Developing Connectors for Data Stores Without SQL.

✓ **Day Three Checkpoint – Verify Your Progress**

- QSTypeInfoMetadataSource returns the data types appropriate for your data store (TODO #7)

- QSMetadataHelper constructor loads the tables defined in your data store
- QSMetadataHelper::GetNextTable() returns the next table from your data store
- MakeNewMetadataTable() returns null for the metadata sources handled by the metadata helper
- The test tool can retrieve the table list using SQLTables (ODBC Test on Windows; the tables command in iodbctest on Linux/Unix and macOS)
- The test tool can retrieve type information using SQLGetTypeInfo (the datatypes command in iodbctest on Linux/Unix and macOS)

Summary – Day Three

At this point, you have completed the following tasks:

- Learned how ODBC catalog functions describe the structure of your data store to applications.
- Implemented QSTypeInfoMetadataSource to return the data types supported by your data store.
- Modified the metadata helper so the Product can generate table, catalog, schema, table type, and column metadata for your data store.

Your custom ODBC connector can now return type metadata. You can use an ODBC-enabled application to connect to your connector and retrieve type metadata from within your data store.

Day Four: Data Retrieval

Today's goal is to enable data retrieval from within the connector. The instructions in this chapter apply to all platforms.

We will discuss:

- Opening a table defined within your data store
- Retrieving the column information for the table
- Retrieving data

LEARNING OBJECTIVES

By the end of this section, you will be able to:

- Open tables in response to Simba SQL Engine requests
- Validate catalog, schema, and table names in OpenTable()
- Define the columns of your tables in terms of ODBC SQL types
- Implement row navigation and data retrieval in QSTable
- Release data store resources when data retrieval completes
- Test data retrieval by executing SQL queries against your connector

Enable Data Retrieval

TODO #8: Open A Table

QSDataEngine::OpenTable() is the entry point where the Simba SQL Engine requests that tables involved in the query be opened. It is called during the preparation of a SQL statement.

> **Note:** The SQL Engine component of the Product allows applications to execute SQL commands on data stores that are not SQL-capable.

Open the file `QSDataEngine.cpp` and navigate to the line **TODO #8: Open A Table** to go to the relevant section of code. (On Windows, you can double-click the TODO #8 message in the Output window.)

QSTable is an implementation of DSIExtSimpleResultSet, an abstract class that provides basic forward-only result set traversal. The main role of QSTable is to translate the stored data from your native data format into SQL Data types.

The QuickStart sample connector is implemented for Tabbed Unicode Files. The sample connector translates the text from UTF16-LE strings into the SQL Data types defined for each column within the configuration dialog.

In the QuickStart connector, QSTable uses a TabbedUnicodeFileReader, which provides an interface to navigate between lines within a Unicode text file. This class preprocesses each row in the file to determine the starting file offset of each column in the row. Its GetData method takes a columnIndex and uses it to calculate the exact position in the file where the column's data resides. The method repositions the file and retrieves the data as if from a byte-buffer. See *Appendix E: Data Retrieval* for a brief overview of data retrieval.

The following sections explain how to implement data retrieval in your custom ODBC connector.

Modify the OpenTable Method

The `QSDataEngine::OpenTable()` method is called during the preparation of a SQL statement. Modify this method to check that the supplied catalog, schema and table names are valid and correspond to a table defined in your data store. If the inputs are not valid, return null to indicate that the table does not exist. If the inputs are valid, return a new instance of QSTable.

Modify QSTable

This section tells you how to modify QSTable so that it can work with your data store.

Return the catalog, schema and table names for your table

Make the following changes to your custom ODBC connector so that it can work with your data store:

- `QSTable::QSTable()`: Modify the constructor to take in the catalog, schema and table names and save them in member variables.
- `QSTable::GetCatalogName()`: Returns QS_CATALOG.
- `QSTable::GetSchemaName()`: Returns simba_wstring() (because it does not support schemas).
- `QSTable::GetTableName()`: Returns m_tableName.

Return the columns defined for your table

Modify **QSTable::InitializeColumns()** so that, for each column defined in the table, you define a DSIResultSetColumn in terms of SQL types.

Example: pseudo code for a custom QSTable::InitializeColumns()

```
AutoPtr<DSIResultSetColumns> columns;
// Get all column information from your data store for the table
For Each Defined Column
{
    AutoPtr<DSIColumnMetadata> columnMetadata(
        new DSIColumnMetadata());
    columnMetadata->m_catalogName = m_catalogName;
    columnMetadata->m_schemaName = m_schemaName;
```

```
    columnMetadata->m_tableName = m_tableName;
    columnMetadata->m_name = //column name
    columnMetadata->m_label = //localized column name
    columnMetadata->m_unnamed = false;
    columnMetadata->m_charOrBinarySize = //the length in bytes
    columnMetadata->m_nullable = DSI_NULLABLE;

    // Change the first parameter of this method to the SQL
    // type that maps to your data store type.
    SqlTypeMetadata* sqlTypeMetadata =
        SqlTypeMetadataFactory::MakeNewSqlTypeMetadata(
            SQL_WVARCHAR, TDW_BUFFER_OWNED);

    columns->AddColumn(
        new DSIResultSetColumn(
            sqlTypeMetadata,
            columnMetadata.Detach()));
}
m_columns.Attach(columns.Detach());
```

SEN SDK 10.4:

In SEN SDK 10.4, column metadata access is const-correct. `DSIResultSetColumns::AddColumn()` now takes a const IColumn* (an AddColumn(AutoPtr<const IColumn>) overload is also available), and `IColumns::GetColumn()` and IColumn-returning accessors now return const IColumn*. The code above continues to compile because a freshly constructed DSIResultSetColumn* converts to const IColumn* implicitly. However, if your driver overrides any method that previously returned a mutable IColumn*/IColumns* (for example a custom metadata source or column factory), update the override to return the const-qualified type.

Implement Navigation

The methods listed below are responsible for navigating a data structure containing information about one table in the data store, and retrieving data from the table. Modify these methods for your data store:

- `QSTable::MoveToBeforeFirstRow()`
- `QSTable::MoveToNextRow()`
- `TabbedUnicodeFileReader::GetData()`

In your custom implementation:

- It is best to implement a streaming interface for the data in the table within your data store.
- Provide the ability to navigate forward from one table row to the next.
- Provide the ability to navigate across columns within the row.
- Provide the ability to read the data associated with the current row and column combination.

Modify the DoCloseCursor Method

`QSTable::DoCloseCursor()` is a callback method called from Simba SQL Engine to indicate that data retrieval has completed, and that you may now do any tasks related to closing the connection to your data store.

✓ **Day Four Checkpoint – Verify Your Progress**

- QSDataEngine::OpenTable() validates the catalog, schema, and table names and returns a QSTable instance (TODO #8)
- QSTable returns the catalog, schema, and table names for your tables
- QSTable::InitializeColumns() defines a DSIResultSetColumn for each column in terms of SQL types
- MoveToBeforeFirstRow(), MoveToNextRow(), and GetData() navigate and retrieve your data
- DoCloseCursor() performs any cleanup needed when data retrieval completes
- SELECT queries return proper result sets in your test tool
- SQLTables() and SQLColumns() return the correct metadata

Summary – Day Four

At this point, you have completed the following tasks:

- Modified OpenTable() to open the tables in your data store that are involved in a query.
- Defined the columns of your tables in terms of ODBC SQL types.
- Implemented row navigation and data retrieval so the Simba SQL Engine can read data from your data store.

You can now execute queries and retrieve data from your data store, and see the rest of the metadata from your data store.

Windows

You should be able to:

- Run `SQLTables()` and `SQLColumns()` from within ODBCTest32.exe (Unicode) and see the correct metadata returned.
- Execute queries from any ODBC-enabled application such as Microsoft Excel, Microsoft Access, Microsoft SQL Server or Crystal Reports and see the results returned from your data store.

Linux/Unix and macOS

Lists of catalogs, schemas, tables and types are available using the qualifiers, owners, tables and types commands in the iodbctest utility. You can use any ODBC-enabled application to execute queries and see the results returned from your data store.

Day Five: Productization

Today's goal is to start productizing and rebranding your custom ODBC connector. Additionally, you can also start localizing your connector error messages; refer to Developing Connectors for Data Stores Without SQL for more details. The instructions in this chapter apply to all platforms; platform-specific steps are identified with a platform heading.

LEARNING OBJECTIVES

By the end of this section, you will be able to:

- Configure custom error messages for your connector
- Set vendor branding for all error messages
- Update the vendor name in a Tableau Datasource Connection (TDC) file
- Rename files and classes for production deployment
- Create a connector configuration dialog
- Finalize the connector for customer deployment

Rebrand Error Messages

TODO #9: Register the QSMessages.xml file for handling by DSIMessageSource.

TODO #10: Set the vendor name, which will be prepended to error messages.

Error messages sent by the connector are visible to applications and customers. In the QuickStart sample connector, error messages are branded with QuickStart, QS, and Simba. This section explains how to rebrand the error messages to reflect the custom connector name and the company name.

All the error messages used within the DSI implementation are stored in a file called `QSMessages.xml`.

To configure error messages:

1. Rename the `QSMessages.xml` file to reflect the name of your company or your custom ODBC connector.
2. Open the file QSDriver.cpp and navigate to the **TODO #9 Register the QSMessages.xml file for handling by DSIMessageSource** message to go to the relevant section of code.
3. Update the line associated with the TODO to match the new name of the `QSMessages.xml` file.
4. Open the `QSMessages.xml` file and change all instances of the following items:
   - Change the letters QS to an appropriate two-letter abbreviation.
   - Change the word QuickStart to an appropriate name for your custom connector.

5. For each exception thrown within the custom DSI implementation, change the parameters to match your custom connector name. This rebrands the error messages to reflect the name of your connector.
6. Open the file QSDriver.cpp and navigate to the **TODO #10 Set the vendor name, which will be prepended to error messages** message to jump to the relevant section of code.
7. The vendor name is prepended to all error messages that are visible to applications. As explained in the code comments, change the vendor name from Simba to an appropriate name for your company. The default vendor name is Simba.

Update the vendor name in the Tableau Datasource Connection (TDC) file

A TDC file contains configuration information that will be applied to any Tableau connection that matches the database vendor name and connector name described in the TDC file.

To set the vendor name for your custom ODBC connector:

1. Ensure you have set your vendor name as described in TODO #10.
2. In the class that extends DSIConnection (QSConnection in our sample QuickStart Connector), set the property DSI_CONN_DBMS_NAME to your vendor name. Example:

   ```
   SetProperty(DSI_CONN_DBMS_NAME, AttributeData::Create<ATTR_WSTRING>
   ("YourVendorName"));
   ```

3. Set the vendor name in the TDC file by following the instructions on Tableau's website.

SEN SDK 10.4:

In SEN SDK 10.4, the pointer-returning named factories on AttributeData – including `MakeNewWStringAttributeData()` were removed. Create attribute values with the templated value factory `AttributeData::Create<AttributeType>(...)` instead (for example, `AttributeData::Create<ATTR_WSTRING>("YourVendorName")`). Property setters such as `SetProperty()` now take an AttributeData by value rather than an owned AttributeData* pointer. If you are upgrading a driver written for 10.3, replace every `MakeNew*AttributeData()` call accordingly.

By default, the value of the DSI_CONN_DBMS_NAME property is TEXT. These steps allow Tableau to match the vendor name in the TDC file with the associated `SQLGetInfo()` property it queries the connector for.

Rebrand the Custom ODBC Connector (Adding Finishing Touches)

All the TODOs in the QuickStart sample connector project are finished, and the custom connector is rebranded and retrieving data from your data store. To complete the custom connector, add the following functionality:

1. Rename all files and classes in the project to have the two-letter abbreviation chosen as part of TODO #9.

2. Create a connector configuration dialog. This dialog is presented to the user when they create a new ODBC DSN or configure an existing one.

Windows

The C++ SimbaEngine QuickStart driver project contains an example ODBC configuration dialog that you can look at, as an example. You can find the source under the Setup folder within the Simba SDK QuickStart driver project.

To see the driver configuration dialog that you created, run the ODBC Data Source Administrator: open the Control Panel, select **Administrative Tools**, and then select **Data Sources (ODBC)**. If your Control Panel is set to view by category, then Administrative Tools is located under System and Security.

> **Important:** If you are using 64-bit Windows with 32-bit applications, you must use the 32-bit ODBC Data Source Administrator, located at C:\WINDOWS\SysWOW64\odbcad32.exe. See Appendix A: ODBC Data Source Administrator on Windows 32-Bit vs. 64-Bit for details.

Linux/Unix and macOS

Creating a connector configuration dialog is possible on Linux and Unix platforms. However, note that the QuickStart connector project for Linux and Unix platforms does not contain an example ODBC configuration dialog.

✓ **Day Five Checkpoint – Verify Your Progress**

- QSMessages.xml is renamed and registered with DSIMessageSource (TODO #9)
- All instances of the QS abbreviation and the QuickStart name are replaced in the messages file
- Exceptions thrown in the DSI implementation are rebranded to match
- The vendor name prepended to error messages is set to your company name (TODO #10)
- All files and classes in the project are renamed with your two-letter abbreviation
- A connector configuration dialog is created (Windows)
- Connector builds, connects, and returns rebranded error messages

Summary – Day Five

At this point, you have completed the following tasks:

- Rebranded the error messages, including the messages XML file and the vendor name, to reflect your custom connector and company name.
- Renamed the project files and classes for production deployment.
- Added the finishing touches, including a connector configuration dialog on Windows.

Conclusion

You have written a custom ODBC connector that can be used by ODBC-enabled applications to query and retrieve data from a custom data store. The custom ODBC connector is renamed and rebranded for your company and product. You now have a read-only connector that connects to your data store.

Reference

This section contains more information that you may find useful when developing your custom ODBC connector. It is organized into the following appendices:

- Appendix A: ODBC Data Source Administrator on Windows 32-Bit vs. 64-Bit
- Appendix B: Bitness and the Windows Registry
- Appendix C: Driver Managers (Linux/Unix and macOS)
- Appendix D: Locating the Configuration Files (Linux/Unix and macOS)
- Appendix E: Data Retrieval
- Appendix F: C++ Server Configuration
- Appendix G: How to Add Schema Support
- Appendix H: Install the Evaluation License
- Appendix I: Troubleshooting (Linux/Unix and macOS)
- Appendix J: Third-Party Licenses

Appendix A: ODBC Data Source Administrator on Windows 32-Bit vs. 64-Bit

On a 64-bit Windows system, you can execute 64-bit and 32-bit applications transparently. Currently, many applications are available in 32-bit versions only. Running 32-bit applications on 64-bit operating systems is common.

> **Note:**
> Microsoft Excel is available in 32-bit and 64-bit versions.

In a single running process, all of the code must be either 64-bit or 32-bit. 64-bit applications can only load 64-bit ODBC drivers and 32-bit applications can only load 32-bit ODBC drivers.

On a 64-bit Windows system, the ODBC Data Source Administrator that you access through the Control Panel by default configures data sources for 64-bit applications. However, you must use the 32-bit version of the ODBC Data Source Administrator to configure data sources for 32-bit applications.

PROBLEM: You cannot access the 32-bit ODBC Data Source Administrator from the Start menu or Control Panel in 64-bit Windows.

SOLUTION: To create new 32-bit data sources or modify existing ones on 64-bit Windows you must run `C:\WINDOWS\SysWOW64\odbcad32.exe`. Create a shortcut to the 32-bit ODBC Data Source Administrator on your Desktop or Start menu if you configure 32-bit data sources frequently.

Appendix B: Bitness and the Windows Registry

A 64-bit application cannot use a 32-bit ODBC driver, and vice versa. In the Registry on computers running 64-bit Windows operating systems, system-wide information about 64-bit ODBC drivers is stored in the **HKEY_LOCAL_MACHINE/SOFTWARE/ODBC** subkey and system-wide information about 32-bit ODBC drivers is stored in the **HKEY_LOCAL_MACHINE/SOFTWARE/WOW6432NODE/ODBC** subkey.

For 32-bit applications running on a 64-bit computer, 32-bit data sources appear the same as if the application was running on a 32-bit computer.

On computers running 32-bit Windows operating systems, system-wide information about ODBC drivers is stored in the **HKEY_LOCAL_MACHINE/SOFTWARE/ODBC** subkey. 32-bit Windows operating systems cannot run 64-bit applications or drivers.

The 32- and 64-bit ODBC Data Source Administrators discussed in the previous section use information about drivers stored in the respective Windows Registry keys.

The ODBCINST.INI Subkey

To define a name and set the location of drivers, the ODBC Data Source Administrator matching the bitness of the operating system uses definitions in the subkey **HKEY_LOCAL_MACHINE/SOFTWARE/ODBC/ODBCINST.INI** and the 32-bit ODBC Data Source Administrator on 64-bit Windows uses Registry keys created in **HKEY_LOCAL_MACHINE/SOFTWARE/WOW6432NODE/ODBC/ODBCINST.INI**.

The ODBCINST.INI subkey contains a subkey for each driver. The subkey for each driver includes the following string values:

- Driver provides the path to the driver DLL
- Setup provides the location of the driver setup DLL
- Description briefly describes the driver

The ODBCINST.INI subkey also contains a subkey named ODBC Drivers that contains string values to match the name of each driver subkey. Each string value is the text Installed.

For more details on information in the Windows Registry contained in the ODBCINST.INI subkey, see the MSDN topic Registry Entries for ODBC Components at https://msdn.microsoft.com/en-us/library/ms714818%28v=vs.85%29.aspx.

The ODBC.INI Subkey

To connect a driver to a database, the ODBC Data Source Administrator matching the bitness of the operating system uses definitions in the subkey **HKEY_LOCAL_MACHINE/SOFTWARE/ODBC/ODBC.INI** and the 32-bit ODBC Data Source Administrator on 64-bit Windows uses Registry keys created in **HKEY_LOCAL_MACHINE/SOFTWARE/WOW6432NODE/ODBC/ODBC.INI**.

The ODBC.INI subkey contains a subkey for each Data Source Name (DSN). The subkey for each DSN includes the following string values:

- **Driver** defines the name of the driver to use for connecting to the database, as set in the ODBCINST.INI subkey. The Driver value can also set a file path to the driver DLL.
- **DBF** defines the path to the database to which the DSN creates a connection.
- **Description** briefly describes the DSN.

The ODBC.INI subkey also contains a subkey named ODBC Data Sources that contains string values to match the name of each DSN subkey. Each string value is the name of the driver that the DSN uses.

For more details on information in the Windows Registry contained in the ODBC.INI subkey, see the MSDN topic Registry Entries for Data Sources at https://msdn.microsoft.com/en-us/library/ms712603%28v=vs.85%29.aspx.

Appendix C: Driver Managers (Linux/Unix and macOS)

Most Linux, Unix, and macOS installations do not come with a driver manager as part of the operating system. You must install your own driver manager before you can compile and test your connector under Linux, Unix, or macOS. The following driver managers are supported by the Product:

Driver Manager	Download Location and Product Support
iODBC	www.iodbc.org – Linux and Unix SDK, macOS SDK
unixODBC	www.unixodbc.org – Linux and Unix SDK

Note:
This document uses the iODBC driver manager as an example, because it is supported by the Product on all Linux, Unix, and macOS platforms. It also contains an ODBC test utility.

How do I know where my driver manager is installed?

If you did not install the driver manager yourself, you can look for it in typical installation directories. The driver manager must be installed to a directory that is on the library path:

- LD_LIBRARY_PATH on most Linux platforms
- DYLD_LIBRARY_PATH on macOS

The iODBC driver manager is often installed to /usr/lib or /usr/local/lib. If you do not know where your driver manager is installed, try searching those directories for libraries containing the name libiodb.

Appendix D: Locating the Configuration Files (Linux/Unix and macOS)

The driver manager loads configuration information from the first odbc.ini and odbcinst.ini files that it finds. The Product loads connector-specific configuration information from the first `.simba.QuickStart.ini` file that it finds. Having multiple copies of these configuration files is not

uncommon on development or customer machines, but can lead to confusion when a connector is not loaded as expected.

Important: Though we recommend only having one copy of each of the configuration files on your or your customer's machine, multiple versions may exist in different locations. The driver manager and the Product will load the first version of the file that they find.

Note: On macOS, the Product installer configures the odbc.ini and odbcinst.ini files with information for the sample connectors, making it easy for you to get started with your custom connector. However, there are many different locations where the configuration files can be stored, and it is possible for multiple versions of these files to exist on a customer's machine.

Locating the odbc.ini File

Linux/Unix

The driver manager looks for this file in the following locations, in the order specified:

1. If the ODBCINI environment variable is set, the driver manager looks in the directory specified by this variable. The environment variable is different for each driver manager, as shown in Using Environment Variables to Specify the Location of the Configuration Files.
2. If there is no variable set, or if no file is found in that location, the driver manager looks in the user's home directory, ~/.

   Note: In this directory the file must have a preceding dot, for example: ~/.odbc.ini
3. The driver manager looks in the /etc directory, for example /etc/odbc.ini.

   Note: There is no dot in front of the odbc.ini file in the /etc directory.

macOS

The driver manager looks for this file in the following locations, in the order specified:

1. If the ODBCINI environment variable is set, the driver manager looks in the directory specified by this variable.
2. If there is no variable set, or if no file is found in that location, the driver manager looks in the user's home directory, ~/.

   Note:
   In this directory the file must have a preceding dot, for example: ~/.odbc.ini
3. The driver manager looks in the user-specific Library directory, ~/Library/ODBC.
4. The driver manager looks in the /etc directory, for example /etc/odbc.ini.

   Note:
   There is no dot in front of the odbc.ini file in the /etc directory.
5. The driver manager looks in the system-wide Library directory, /Library/ODBC.

Locating the odbcinst.ini File

Linux/Unix

The driver manager looks for this file in the following locations, in the order specified:

1. If the corresponding environment variable is set, the driver manager looks in the directory specified by this variable. The environment variable is different for each driver manager, as shown in Using Environment Variables to Specify the Location of the Configuration Files.
2. If there is no variable set, or if no file is found in that location, the driver manager looks in the user's home directory.

   Note:
   In this directory the file must have a preceding dot, for example: ~/.odbcinst.ini

3. The driver manager looks in the /etc directory, for example /etc/odbcinst.ini.

   Note:
   There is no dot in front of the odbcinst.ini file in the /etc directory.

macOS

The driver manager looks for this file in the following locations, in the order specified:

1. If the corresponding environment variable is set, the driver manager looks in the directory specified by this variable.
2. If there is no variable set, or if no file is found in that location, the driver manager looks in the user's home directory.

   Note:
   In this directory the file must have a preceding dot, for example: ~/.odbcinst.ini

3. The driver manager looks in the user-specific Library directory, ~/Library/ODBC.
4. The driver manager looks in the /etc directory, for example /etc/odbcinst.ini.

   Note:
   There is no dot in front of the odbcinst.ini file in the /etc directory.

5. The driver manager looks in the system-wide Library directory, /Library/ODBC.

Locating the .simba.QuickStart.ini Configuration File

The Product searches for the .simba.QuickStart.ini file in the following locations, in the specified order (Linux, Unix, and macOS platforms):

1. The path, including the file name, specified using the SIMBAINI environment variable.

   Note:
   The name of this environment variable can be rebranded.

2. The connector directory, as a non-hidden .ini file.
3. The directory that the client application is launched from.
4. In $HOME, as a hidden .ini file.
5. In /etc/ as a non-hidden .ini file.

Using Environment Variables to Specify the Location of the Configuration Files

The location of configuration files is determined by environment variables, as shown below:

Environment Variable	File
ODBCINI	Specifies the full path (including the file name) of the odbc.ini file for the iODBC and unixODBC driver managers.
ODBCINSTINI	Specifies the full path (including the file name) of the odbcinst.ini file for the iODBC driver manager.
ODBCSYSINI	Specifies the directory containing the odbcinst.ini file for the unixODBC driver manager (Linux and Unix platforms).
SIMBAINI	Specifies the full path (including the file name) of the connector configuration file. By default, this file is called simba.drivername.ini. Note: the name of this environment variable can be rebranded.

For example, these environment variables could be set as shown below:

```
export ODBCINI=/usr/local/odbc/myodbc.ini
export ODBCINSTINI=/usr/local/odbc/myodbcinst.ini
export ODBCSYSINI=/usr/local/odbc
```

If the environment variables are not set, the driver manager assumes that the configuration files exist in the user's home directory using the default file names .odbc.ini and .odbcinst.ini.

Appendix E: Data Retrieval

In the Data Store Interface (DSI), the following methods perform the actual task of retrieving data from your data store:

- Each MetadataSource implementation of `GetMetadata()`
- `TabbedUnicodeFileReader::GetData()` (called via `QSTable::GetData()`)

Both methods provide a way to uniquely identify a column within the current row. For MetadataSource, the Simba SQL Engine passes in a unique column tag (see DSIOutputMetadataColumnTag). For QSTable, the Simba SQL Engine will pass in the column index.

SEN SDK 10.4:

In SEN SDK 10.4, the data-retrieval contract was modernized. The legacy bool overloads that took a SqlData* pointer and a signed offset – bool `RetrieveData(..., SqlData* in_data, simba_signed_native in_offset, simba_signed_native in_maxSize)` and the matching bool `GetMetadata(DSIOutputMetadataColumnTag, SqlData*, ...)` – are deprecated. The new primary overloads take a SqlData& reference and a simba_uint64 in_offset, and they return a simba_int64 giving the number of bytes remaining (or SQL_NO_TOTAL) instead of a bool:

```
simba_int64 RetrieveData(SqlData& out_data, simba_uint64 in_offset, simba_signed_native in_maxSize)
```

```
simba_int64 GetMetadata(DSIOutputMetadataColumnTag in_tag, SqlData& out_data, simba_uint64 in_offset, simba_signed_native in_maxSize)
```

The examples in this appendix use the legacy pointer form (`in_data->...`) for continuity with existing 10.3 sample code, which still compiles against the deprecated overloads in 10.4. When writing new 10.4 code, prefer the reference/byte-count overloads and access the buffer as `in_data.GetBuffer()`. Length and offset parameters throughout the SDK were also widened to size_t/simba_uint64.

In addition, both methods accept the following three parameters:

in_data

The SQLData into which you must copy the value of your cell. This class is a wrapper around a buffer managed by the Simba SQL Engine. To access the buffer, you call its `GetBuffer()` method. The data you copy into the buffer must be formatted as a SQL Type (see https://msdn.microsoft.com/en-us/library/ms710150%28VS.85%29.aspx for a list of data types and definitions). Therefore, if your data is not stored as SQL Types, you will need to write code to convert from your native format.

The type of this parameter is governed by the metadata for the column that is returned by the class. Thus, if you set the SQL Type of column 1 in `QSTable::InitializeColumns()` to SQL_INTEGER, then when `GetData()` is called for column 1, you will be passed a SQLData that wraps an int data type. For MetadataSource, the type is associated with the column tag (see DSIOutputMetadataColumnTag.h).

Example: If SqlData was of type SQL_INTEGER:

```
simba_int32 value = 5;
// This is one way
memcpy(in_data->GetBuffer(), &value, sizeof(simba_int32));
// This is another way; both work equally well
*reinterpret_cast<simba_int32*>(in_data->GetBuffer()) = 5;
```

Important:
When working with variable length data, for example character or binary data, you must call `SetLength()` before calling `GetBuffer()`. Not doing so may result in a heap violation. See QSTypeUtilities.h for an example on how to handle character or binary data.

in_offset

Character, wide character and binary data types can be retrieved in parts. This value specifies where, in the current column, the value should be copied from. The value is usually `0`.

in_maxSize

The maximum size (in bytes) that can be copied into the in_data parameter. For character or binary data, copying data that is greater than this size can result in a data truncation warning or a heap violation.

SqlData Types

SqlData objects represent the SQL types and encapsulate the data in a buffer. To get the underlying SQL type that a SqlData object represents, use `GetMetadata()->GetSqlType()`. This retrieves the associated `SQL_*` type.

For information on how SQL types map to C++ types, see SQL Data Types in Developing Connectors for Data Stores Without SQL.

Fixed-Length Types

The structures used to store the fixed-length data types represented by SqlData objects are:

SQL_BIT	SQL_INTERVAL_MINUTE
SQL_DATE	SQL_INTERVAL_MINUTE_TO_SECOND
SQL_DECIMAL	SQL_INTERVAL_MONTH
SQL_DOUBLE	SQL_INTERVAL_SECOND
SQL_FLOAT	SQL_INTERVAL_YEAR
SQL_INTERVAL_DAY	SQL_INTERVAL_YEAR_TO_MONTH
SQL_INTERVAL_DAY_TO_HOUR	SQL_NUMERIC
SQL_INTERVAL_DAY_TO_MINUTE	SQL_REAL
SQL_INTERVAL_DAY_TO_SECOND	SQL_BIGINT
SQL_INTERVAL_HOUR	SQL_INTEGER
SQL_INTERVAL_HOUR_TO_MINUTE	SQL_SMALLINT
SQL_INTERVAL_HOUR_TO_SECOND	SQL_TINYINT
SQL_TYPE_DATE	SQL_TYPE_TIME
SQL_TYPE_TIMESTAMP	

More information on Date, Time and DateTime types: The associated SQL types for date, time, and datetime are SQL_TYPE_DATE, SQL_TYPE_TIME, and SQL_TYPE_TIMESTAMP. Please note that SQL_TIME, SQL_DATE, and SQL_TIMESTAMP are ODBC 2.x types while the SQL_TYPE_* types are

ODBC 3.x types, so you should be sure to use the SQL_TYPE_* types since you are developing an ODBC 3.x connector.

Simple Fixed-Length Data Example: For a SQL_INTEGER, the SQLData will contain a simba_int32 which you must copy your integer value into. The example below illustrates how this might be achieved.

```
switch (in_data->GetMetadata()->GetSqlType())
{
    case SQL_INTEGER:
    {
        simba_int32 value = 1234;
        *reinterpret_cast<simba_int32*>(in_data->GetBuffer()) = value;
    }
}
```

Variable-Length Types

The following variable-length data types are stored in buffers and represented by SqlData objects:

SQL_BINARY	SQL_LONGVARBINARY	SQL_VARBINARY
SQL_CHAR	SQL_LONGVARCHAR	SQL_VARCHAR
SQL_WCHAR	SQL_WLONGVARCHAR	SQL_WVARCHAR

You may find that DSITypeUtilities::OutputWVarCharStringData and OutputVarCharStringData are useful for setting character data.

Simple Variable-Length Data Example: The SQL_CHAR example below illustrates how the type utilities might be used while the SQL_VARCHAR example shows a simple example using memcpy. In practice, SQL_CHAR, SQL_VARCHAR and SQL_LONGVARCHAR will not need separate cases to handle them and there will also be other considerations such as having to deal with offsets into the data.

```
switch (in_data->GetMetadata()->GetSqlType())
{
    case SQL_CHAR:
    {
        simba_string stdString("Hello");
        return DSITypeUtilities::OutputVarCharStringData(
            &stdString,
            in_data,
            in_offset,
            in_maxSize);
```

```
    }
    case SQL_VARCHAR:
    {
        simba_string stdString("Hello");
        simba_uint32 size = stdString.size();
        in_data->SetLength(size);
        memcpy(in_data->GetBuffer(), stdString, size);
        return false;
    }
}
```

Data Conversion in Practice

In the QuickStart example, when `GetData()` is called the values are read from the tabbed Unicode file (in `TabbedUnicodeFileReader::GetData`), converted to simba_wstrings (in `QSTable::ReadWholeColumnAsString`) and then converted to the requested SQL data type (in `QSTable::ConvertData`). This works well because the data source is a text file and a good cross-platform example.

For your data source, if you're already getting data of the correct type–integers, for example–then ideally you should skip the conversion to strings so you can achieve better performance. Be aware of which data types map to which SQL Types, as well as how to represent them in the expected format. Then you can set the buffer in an appropriate manner.

NULL Values

To represent a null value, directly set the SqlData object as null: `in_data->SetNull(true);`

Appendix F: How to Add Schema Support

Some applications require support for schemas (note that Microsoft Excel does not require schema support). If the data store supports schemas, the connector can provide access to them. To support schemas, a connector must handle DSI_SCHEMAONLY_METADATA.

To add schema support to a connector:

1. By default, `QSConnection::SetConnectionPropertyValues()` disables schema support via `DSIPropertyUtilities::SetSchemaSupport()`. Change this value to true to enable schema support.
2. Make the following additional changes:
    - `QSMetadataHelper::GetNextTable()` – In the QuickStart sample connector, a blank schema is returned because schema support is not enabled by default. To enable schema support, return the schema in the Identifier to allow the Product to open the correct table.
    - `QSDataEngine::OpenTable()` – Modify this method to verify the given schema and return the correct table for the given catalog, schema and table name.

- `QSTable::GetSchemaName()` — Modify this method to return the schema to which the table belongs.

Appendix G: Install the Evaluation License

You can use the Product for 30 days after installing the evaluation license. The evaluation license is emailed to the person who registered the product.

Typically, you use the Product to create your custom ODBC connector, then use a test ODBC-enabled application to retrieve data using the connector.

Linux/Unix and macOS

To license the Product on Unix, Linux, and macOS, save the license file under the $HOME directory, either as a hidden or a non-hidden file. For example:

- `/home/<user_id>/SimbaEngineSDK.lic` for a non-hidden file
- `/home/<user_id>/.SimbaEngineSDK.lic` for a hidden file

Appendix H: Troubleshooting (Linux/Unix and macOS)

This section contains solutions to common problems.

Specified Driver Could Not Be Loaded

On Linux, Unix, and macOS platforms, the error Specified Driver Could Not Be Loaded may be returned when you try to connect to a data source. Here are some of the reasons the driver manager cannot load the specified connector:

- You are using a 64-bit driver manager, but have 32-bit paths defined in the odbc.ini or odbcinst.ini files.
- You define the path to the debug version of the connector, but you built the non-debug version.

Unable to Connect to the Connector

In the `odbc.ini` and `odbcinst.ini` files, be sure there are no characters such as spaces or tabs at the end of a value. This can cause failure to connect.

Is My Driver Manager 32- or 64-bit?

You can use the file command. For example, this version of iodbctest (and therefore the iODBC driver manager) is 64-bit:

```
/usr/local/bin> file iodbctest
iodbctest: ELF 64-bit LSB executable, x86-64, version .....
```

See also Testing and Troubleshooting in the guide *Developing Connectors for Data Stores Without SQL*.

Appendix J: Third-Party Licenses

The Simba SDK includes software developed by third parties. This section provides license information for the third-party components included in the SDK. For complete license information, please refer to the third-party-licenses.txt file that is packaged with the Simba SDK software.

> **Note:**
> The third-party-licenses.txt file is in the [INSTALLDIR]\SimbaEngineSDK\10.4\Docs directory and contains the complete text of all third-party licenses applicable to the SDK.

fast_float License

The Simba SDK uses the fast_float library for high-performance floating-point number parsing.

Apache License, Version 2.0

Copyright © fast_float authors. All rights reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at:
http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

The fast_float library is also available under the MIT License. The MIT License text is as follows:

MIT License

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

ICU License (ICU 1.8.1 and later)

The Simba SDK uses the International Components for Unicode (ICU) library for Unicode support and internationalization services.

COPYRIGHT AND PERMISSION NOTICE

Copyright © 1995-2026 International Business Machines Corporation and others. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

OpenSSL License

The Simba SDK may use OpenSSL for SSL/TLS encryption support in server configurations.

Copyright © 1998-2026 The OpenSSL Project. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- All advertising materials mentioning features or use of this software must display the following acknowledgment: "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
- The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org.
- Products derived from this software may not be called "OpenSSL" nor may "OpenSSL" appear in their names without prior written permission of the OpenSSL Project.

- Redistributions of any form whatsoever must retain the following acknowledgment: "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/)"

THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Original SSLeay License

The OpenSSL toolkit includes cryptographic software written by Eric Young (eay@cryptsoft.com) and software written by Tim Hudson (tjh@cryptsoft.com).

Copyright © 1995-1998 Eric Young (eay@cryptsoft.com). All rights reserved.

This package is an SSL implementation written by Eric Young (eay@cryptsoft.com). The implementation was written so as to conform with Netscape's SSL.

This library is free for commercial and non-commercial use as long as the following conditions are adhered to. The following conditions apply to all code found in this distribution, be it the RC4, RSA, lhash, DES, etc., code; not just the SSL code. The SSL documentation included with this distribution is covered by the same copyright terms except that the holder is Tim Hudson (tjh@cryptsoft.com).

Copyright remains Eric Young's, and as such any Copyright notices in the code are not to be removed. If this package is used in a product, Eric Young should be given attribution as the author of the parts of the library used. This can be in the form of a textual message at program startup or in documentation (online or textual) provided with the package.

libexpat License

The Simba SDK uses the Expat XML parser library (libexpat) for XML parsing functionality.

MIT License

Copyright © 1998-2026 Thai Open Source Software Center Ltd and Clark Cooper

Copyright © 2001-2026 Expat maintainers

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Third-Party Components Summary

The following table summarizes the third-party components included in the Simba SDK:

Component	Purpose and License Type
fast_float	High-performance floating-point parsing – Apache 2.0 / MIT (dual license)
ICU	Unicode and internationalization support – ICU License
OpenSSL	SSL/TLS encryption – OpenSSL License + SSLeay License
libexpat	XML parsing – MIT License

Important:
When distributing applications built with the Simba SDK, you must include appropriate license notices and attributions for any third-party components that are included in your distribution. Refer to the third-party-licenses.txt file for complete license texts and attribution requirements.

For questions regarding third-party licensing or compliance, please contact insightsoftware at www.insightsoftware.com.

Contact Us

For more information or help using this product, please contact our Technical Support staff. We welcome your questions, comments, and feature requests.

> **Note:**
> To help us assist you, prior to contacting Technical Support please prepare a detailed summary of the Product version and development platform that you are using.

You can contact Technical Support via the insightsoftware Support Community at https://insightsoftware.com/.

Third-Party Trademarks

Simba, the Simba logo, Simba SDK, and Simba Technologies are registered trademarks of Simba Technologies Inc. in Canada, United States and/or other countries. All other trademarks and/or servicemarks are the property of their respective owners.

Kerberos is a trademark of the Massachusetts Institute of Technology (MIT).

Linux is the registered trademark of Linus Torvalds in Canada, United States and/or other countries.

Mac and macOS are trademarks or registered trademarks of Apple, Inc. or its subsidiaries in Canada, United States and/or other countries.

Microsoft, Windows, Visual Studio, Excel, and Access are trademarks or registered trademarks of Microsoft Corporation or its subsidiaries in Canada, United States and/or other countries.

Red Hat, Red Hat Enterprise Linux, and CentOS are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in Canada, United States and/or other countries.

SUSE is a trademark or registered trademark of SUSE LLC or its subsidiaries in Canada, United States and/or other countries.

Ubuntu is a trademark or registered trademark of Canonical Ltd. or its subsidiaries in Canada, United States and/or other countries.

All other trademarks are trademarks of their respective owners.

The licenses for third-party products used in the software (including fast_float, ICU, OpenSSL, SSLeay, and libexpat) are reproduced in Appendix J: Third-Party Licenses. This information is also contained in a third-party-licenses.txt file that is packaged with the software.