ODBC on Linux and UNIX platforms

This document contains all the information you need to get started accessing ODBC data sources on Linux and UNIX platforms. The document provides background information about ODBC and its implementation on Linux and UNIX, describes the unixODBC Driver Manager in detail, and lists some commonly used Linux and UNIX applications that support ODBC.

Contents

What is ODBC?

Open Database Connectivity (ODBC) is a standard software API specification for using database management systems (DBMS). ODBC is independent of programming language, database system, and operating system.

ODBC was created by the SQL Access Group and first released in September 1992. ODBC is based on the Call Level Interface (CLI) specifications from SQL, X/Open (now part of The Open Group), and the ISO/IEC.

The ODBC API is a library of ODBC functions. The ODBC API lets applications that support ODBC work with data in any database for which an ODBC driver is available.

The goal of ODBC is to make it possible to access any data from any application, regardless of which DBMS is handling the data. ODBC achieves this by inserting a middle layer called a database driver between the application and the DBMS. This layer translates the application's data queries into commands that the DBMS understands.

ODBC versions

There are (to date) 5 significant versions of ODBC:

Version	Released	Description
1.0	1993	The first version of ODBC. Only a few ODBC 1.0 applications and drivers still exist on Windows, and none we know of on Linux.
2.0	1994	The second version of ODBC. Small reorganisation of the API. For example, `SQLBindParameter` replaced `SQLSetParam`; core, level 1 and level2 2 conformance changes; new data types. There are still a number of ODBC 2.0 applications and drivers around. On Linux, most ODBC drivers are ODBC 3.*x*, and the few that are still ODBC 2.0 are generally moving to 3.*x*.

		There was also an ODBC 2.5.
3.0	1995	ODBC 3.0 introduced a large number of new APIs and ODBC descriptor handles. Most ODBC drivers on Linux are now ODBC 3.0 and many applications are also 3.0.
3.5*x*	1997	Introduction of Unicode support. Driver-aware connection pooling, which allows an ODBC driver to better estimate the cost of reusing a connection from the pool based on a user's connection settings. Asynchronous connection operation, which enable applications to populate multiple connections in the pool at startup time so that subsequent connection requests can be more efficiently served.
3.8*x*	2009	Driver-specific C data types. These are useful for supporting new DBMS data types that existing C types don't correctly represent. Before version 3.8, ODBC drivers had to use a generic type such as `SQL_C_BINARY` to work with DBMS-specific types, which the application would then need to reconstruct. Streamed output parameters, which enable an application to call `SQLGetData` with a small buffer multiple times to retrieve a large parameter value, reducing the application's memory footprint. We provide a streamed output parameter example for SQL Server in our C samples section.

ODBC components

A basic implementation of ODBC on Linux consists of:

- An application that supports ODBC. This means the application can use the ODBC API to talk to a DBMS.
- The ODBC Driver Manager. The ODBC Driver Manager links between an ODBC application and an ODBC driver. Applications that require ODBC access link with the Driver Manager and make ODBC API calls. The Driver Manager loads the appropriate ODBC driver as a result of these calls. The ODBC Driver Manager also provides other functionality.
- A repository containing a list of installed ODBC drivers and defined ODBC data sources. The ODBC Driver Manager normally looks after these definitions and consults them when applications connect to a data source.
- An ODBC driver. The ODBC driver translates ODBC API calls into something the back-end DBMS understands.

ODBC also includes:

- A cursor library.
- Utilities and APIs to install, remove, and query installed ODBC drivers.
- Data source APIs that allow data sources to be created or managed from within an application. For example, `ConfigDSN`.
- Utility APIs an ODBC driver can use to handle the reading and writing of data source definitions. For example, `SQLGetPrivateProfileString`.
- A GUI and non-GUI ODBC Administrator.
- All the header files required to build ODBC applications.

What is the state of ODBC on Linux?

ODBC on Linux is in a healthy state today. Many applications and interfaces support ODBC and there's a wealth of available ODBC drivers.

The general goal of ODBC on Linux was to:

1. Replicate the ODBC functionality available on Windows so that application authors could write ODBC applications that worked on Windows and Linux or UNIX. This required writing an ODBC Driver Manager.

   For the most part, this has been achieved with unixODBC, which provides a fully ODBC 3.5 compatible Driver Manager. unixODBC provides the full ODBC API and includes the following:

   - All ODBC driver utility functions.
   - ODBC driver installer, uninstaller, and configuration libraries.
   - A GUI administrator.
   - An `odbctest` utility.
   - Development header files.
   - A command line administration utility (`odbcinst`).
   - A command line ODBC application to test data sources and submit SQL to the underlying ODBC driver.
2. Make ODBC drivers available on Linux. There are now a large number of commercial and open-source ODBC drivers for Linux and UNIX platforms.

ODBC Driver Managers

There are two open-source ODBC Driver Managers for Linux and UNIX: unixODBC and iODBC). This document describes the unixODBC Driver Manager as it's the one that's included with most (if not all) Linux distributions and some UNIX distributions.

What does the ODBC Driver Manager do?

The ODBC Driver Manager is the interface between an ODBC application and the ODBC driver. The Driver Manager principally provides the ODBC API so ODBC applications may link with a single shared object and be able to talk to a range of ODBC drivers. For example, an application on Linux links with `libodbc.so` (the main Driver Manager shared object) without having to know at link time which ODBC driver it's going to be using. At run time, the application provides a connection string, which defines the ODBC data source it wants to connect to and this in turn defines the ODBC driver that will handle this data source. The Driver Manager loads the requested ODBC driver (with `dlopen(3)`) and passes all ODBC API calls on to the driver. In this way, an ODBC application can be built and distributed without knowing which ODBC driver it will be using.

However, this is a rather simplistic description of what the Driver Manager does. The ODBC Driver Manager also:

- Controls a repository of installed ODBC drivers. On Linux, this is the file `odbcinst.ini`.

- Controls a repository of defined ODBC data sources. On Linux, these are the files `odbc.ini` and `.odbc.ini`.
- Provides the ODBC driver APIs (`SQLGetPrivateProfileString` and `SQLWritePrivateProfileString`) to read and write ODBC data source attributes.
- Handles `ConfigDSN`, which the ODBC driver exports to configure data sources.
- Provides APIs to install and uninstall ODBC drivers (`SQLInstallDriver`).
- Maps ODBC versions so that an ODBC 2.x application can work with an ODBC 3.x driver and vice versa.
- Maps ODBC states between different versions of ODBC.
- Provides a cursor library for ODBC drivers that only support forward-only cursors.
- Provides `SQLDataSources` and `SQLDrivers` so that an application can find out what ODBC drivers are installed and what ODBC data sources are defined.
- Provides an ODBC Administrator, which lets ODBC driver writers install ODBC drivers and users define ODBC data sources.

ODBC drivers

An ODBC driver exports the ODBC API so that an ODBC application can communicate with a DBMS. Sometimes the ODBC driver is single-tier, which means the driver accesses database files directly. Sometimes the ODBC driver is multi-tier, which means the driver communicates with the DBMS through another layer.

There are a large number of commercial and open-source ODBC drivers available for Linux and UNIX platforms.

Download ODBC Drivers for Oracle, SQL Server, Salesforce, MongoDB, Access, Derby, InterBase & DB2.

ODBC bridges and gateways

An ODBC bridge or gateway provides an ODBC API at one end of the bridge or gateway and a different API at the other end. The most popular API people want to bridge to and from ODBC is JDBC.

ODBC-JDBC gateways

An ODBC-JDBC gateway allows an application that uses the ODBC API to talk to a JDBC driver:

```
application <-> ODBC API <-> JDBC API <-> database
```

An example of this is the Easysoft ODBC-JDBC Gateway.

You would typically use an ODBC-JDBC gateway if you had an existing application that used the ODBC API to access databases, and wanted to use that application to access a database for which there was no ODBC driver available, but a JDBC driver was available.

The ODBC calls your application makes are converted to JDBC calls and passed to the JDBC driver. As far as the JDBC driver is concerned, the ODBC driver is just another JDBC application. As far as the

application is concerned, it's using a normal ODBC driver.

You install an ODBC-JDBC gateway on the same machine as your application, and depending on how the gateway was written, you:

1. Install Java and the JDBC driver on the same machine, and the gateway uses the Java Native Interface (JNI) to load the JDBC driver classes.
2. Install a server process on the same machine as the database, Java, and the JDBC driver. The gateway communicates over your network, converting ODBC calls at the client end through a proprietary interface, and connecting to the server process, which uses JDBC to communicate with the JDBC driver. (In this case, the server process is normally written in Java.)

The first of these configurations is the most popular, probably because:

1. It avoids any proprietary interfaces.
2. Java is available for most platforms.
3. Most JDBC drivers are capable of communication over a network anyway.
4. It avoids any extra services or processes.
5. Nothing has to be installed on the server or database machine.

What may influence your choice of an ODBC-JDBC gateway is:

- The required JDK version.
- JDBC compatibility.
- Compatibility with JDBC types (1–4).
- Transparency.

Some compromises are nearly always inherent in translating the ODBC API to the JDBC API, but these are usually less than you might think, and a good gateway will be very transparent. A common misconception is that adding a bridge between your ODBC application and JDBC driver will introduce a lot of overhead, but you might be surprised at how quick a good gateway can be.

JDBC-ODBC bridges

A JDBC-ODBC bridge is the opposite of an ODBC-JDBC one. A JDBC-ODBC bridge allows a Java application to access an ODBC driver:

```
Java application <-> JDBC <-> ODBC driver <-> database
```

An example of this is the Easysoft JDBC-ODBC Bridge.

You would typically use a JDBC-ODBC bridge if you had an existing Java application that used the JDBC API, and wanted to access a database for which an ODBC driver was available, but a JDBC driver was not.

For instance, you may want to work with a Microsoft Access database from Java, but there is no Microsoft JDBC driver for Microsoft Access.

The JDBC calls your application makes are converted to ODBC calls and passed to the ODBC driver. As far as the Java application is concerned, it's using a normal JDBC driver. As far as the database is concerned, it's being accessed through the normal ODBC driver.

Because ODBC drivers are always written in C (the ODBC API is a C interface), they are built for particular operating systems and architectures. As a result, the most flexible configuration is one where a server process is installed on the machine containing the ODBC driver, and the JDBC side of the bridge communicates with it over the network from the client side where the JDBC driver is installed. Obviously, at the Java application end, Java will already be in use, and so use of the JDBC client end driver at this side of the bridge is not a problem. (In fact, some bridges offer zero installation JDBC access).

JDBC is inherently Unicode, and so a good JDBC-ODBC bridge will convert JDBC calls into the ODBC API wide functions (`SQLxxxW`) and request `SQL_WCHAR` characters from the database if they are available.

What may influence your choice of a JDBC-ODBC Bridge is:

- The type of JDBC driver offered. A type 3, client/server solution, allows Java applications and ODBC drivers to be on separate machines. Not all JDBC-ODBC bridges are like this.
- Support for recent JDBC specifications, but still providing backward compatibility.
- Java 2 Platform Standard Edition (J2SE) compliancy.
- Unicode support. ODBC Unicode support is substantially different from JDBC, so this feature is often missing.
- Remote serving of the JDBC driver — zero installation of JDBC driver.
- Support for JDBC features even when the underlying ODBC driver does not provide similar support (for example, multiple concurrent statements).
- Transparency. The Java application should not know it is really talking to an ODBC driver.

ODBC-ODBC bridges

ODBC-ODBC bridges are mostly used to access an ODBC driver on one platform from another where it is not available. For example, you have got an ODBC driver for database X on Windows, but your application needs to run on Linux where the X ODBC driver is not available. However, since 64-bit Windows was released, a new problem has arisen; you've got a 32-bit application, which you cannot rebuild, but the 32-bit ODBC driver is no longer available or you need to write a new 64-bit application, but only have access to a 32-bit ODBC driver. ODBC-ODBC bridges like the Easysoft ODBC-ODBC Bridge can solve these problems.

The unixODBC ODBC Driver Manager

What is unixODBC?

unixODBC is a project created to provide ODBC on non-Windows platforms. It includes:

- An ODBC Driver Manager, which adheres to the ODBC specification and replicates all the functionality you may be used to in the Microsoft Windows ODBC Driver Manager. (Refer to What does the ODBC Driver Manager do? and Components of ODBC for further information.)

- A collection of open-source ODBC drivers.
- A number of ODBC applications that demonstrate ODBC usage and provide useful functionality. For example, the GUI DataManager, odbctest and isql. **Note** In unixODBC 2.3.0, the GUI components were moved into a new project and must be installed separately if required.

unixODBC is distributed with Red Hat, Debian, Slackware, Ubuntu, Suse, CentOS, and most of the other Linux distributions and is available as source code.

unixODBC is a mature open-source product having made its first beta release in in January 1999. Version 1.0.0 was release in May 1999 and there have been many releases since.

Obtaining, configuring and building unixODBC

Obtaining unixODBC

unixODBC's web site is at www.unixodbc.org. unixODBC also has a SourceForge project at sourceforge.net/projects/unixodbc. You can download RPMs and the source from either site and you can find the latest development release at `ftp.unixodbc.org/pub/unixODBC`.

Note that all Easysoft ODBC drivers for Linux and UNIX platforms come with unixODBC prebuilt.

Configuring and building unixODBC

The unixODBC source distribution is a gzipped tar file. Uncompress and then untar the file. For example:

```
gunzip unixODBC-2.2.12.tar.gz
tar -xvf unixODBC-2.2.12.tar
```

Change into the resultant directory and run:

```
./configure --help
```

which lists all the options configure accepts. The principal ones you need to pay attention to are:

Option	Description
`--prefix`	This defines where you want to install unixODBC. If you don't specify `--prefix`, the default location is `/usr/local`. If you don't want unixODBC all under a single directory, you can use other configure options like `--bindir` or `--sbindir` for finer control.
`--sysconfdir`	This defines where you want unixODBC configuration files to be stored. This defaults to *`prefix`*`/etc`. The configuration files affected are `odbcinst.ini` (where ODBC drivers are defined), the system `odbc.ini` (where system data sources are defined) and `ODBCDataSources` (where system file DSNs are stored).
`--enable-gui`	The default is `yes` if QT is found. If you want to build the GUI ODBC Administrator, odbctest and DataManager set this to `yes` (`--enable-gui=yes`). You need QT libraries and header files to build the GUI components. You should probably also set `--with-x`. If you turn on `--enable-gui`, configure tries to find QT, its libraries, and its header files. If QT is installed in a single place, you can provide a hint to configure by setting the environment `QTDIR` (or `--with-qt-dir`) to point to the top of the directory tree where QT is installed. If QT libraries and header files are installed in separate trees and not

	the default places like `/usr/lib` and `/usr/include`, use `--with-qt-includes=DIR` and `--with-qt-libraries=DIR`. **Note** In unixODBC 2.3.0, the default for `--enable-drivers` changed to `no` and the GUI components were moved into a new project.
`--enable-threads`	The default is `yes` if thread-support is found on your machine. All modern Linux distributions have `pthreads` support in `glibc`, so it's probably best to leave this set to the default value.
`--enable-readline`	The default is `yes` if libreadline and its headers are found on your machine. This principally only affects unixODBC's isql program. If readline support is found then you can edit text entered at the SQL prompt in isql.
`--enable-drivers`	The default is `yes`. Turning on `--enable-drivers` builds all the ODBC drivers included with unixODBC. These include MySQL, Postgres, MiniSQL, and a text file driver. **Note** In unixODBC 2.3.0, the default for `--enable-drivers` changed to `no` and the GUI components were moved into a new project.
`--enable-iconv`	This defaults to `yes` if libiconv and its header files are found on your machine. If you turn on `--enable-iconv`, unixODBC can do Unicode translations.

Note For information about configuring and building unixODBC on 64-bit platforms, refer to 64-bit ODBC.

Where are ODBC drivers defined?

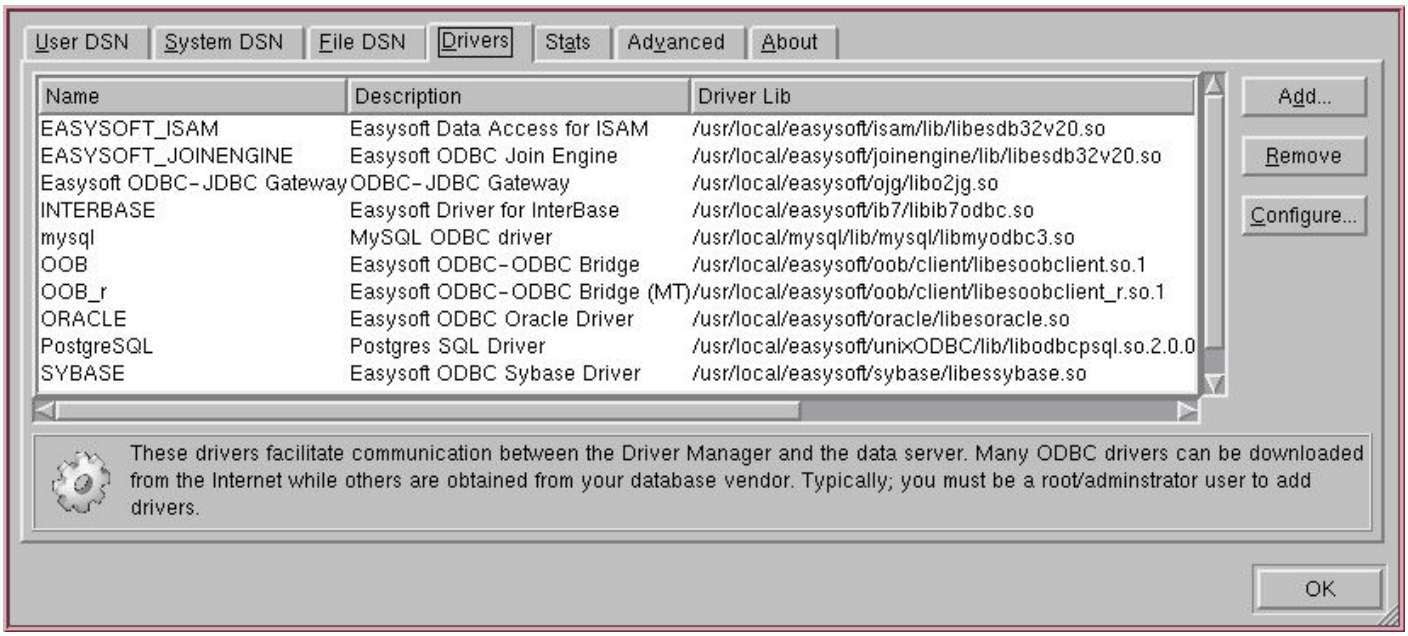
unixODBC defines ODBC drivers in the `odbcinst.ini` file. The location of this file is a configure-time option defined with `--sysconfdir`. If unixODBC is already installed, you can use unixODBC's `odbcinst` program to locate the `odbcinst.ini` file:

```
$ odbcinst -j
unixODBC 2.3.1
DRIVERS............: /etc/odbcinst.ini
SYSTEM DATA SOURCES: /etc/odbc.ini
FILE DATA SOURCES..: /etc/ODBCDataSources
USER DATA SOURCES..: /home/auser/.odbc.ini
SQLULEN Size.......: 4
SQLLEN Size........: 4
SQLSETPOSIROW Size.: 2
```

In this example, drivers are defined in `/etc/odbcinst.ini`.

You can tell unixODBC to look in a different path (to that which it was configured) for the `odbcinst.ini` file and System DSN file (`odbc.ini`) by defining and exporting the `ODBCSYSINI` environment variable. You can tell unixODBC to look in a different file for driver definitions (`odbcinst.ini`, by default) by defining and exporting the `ODBCINSTINI` environment variable.

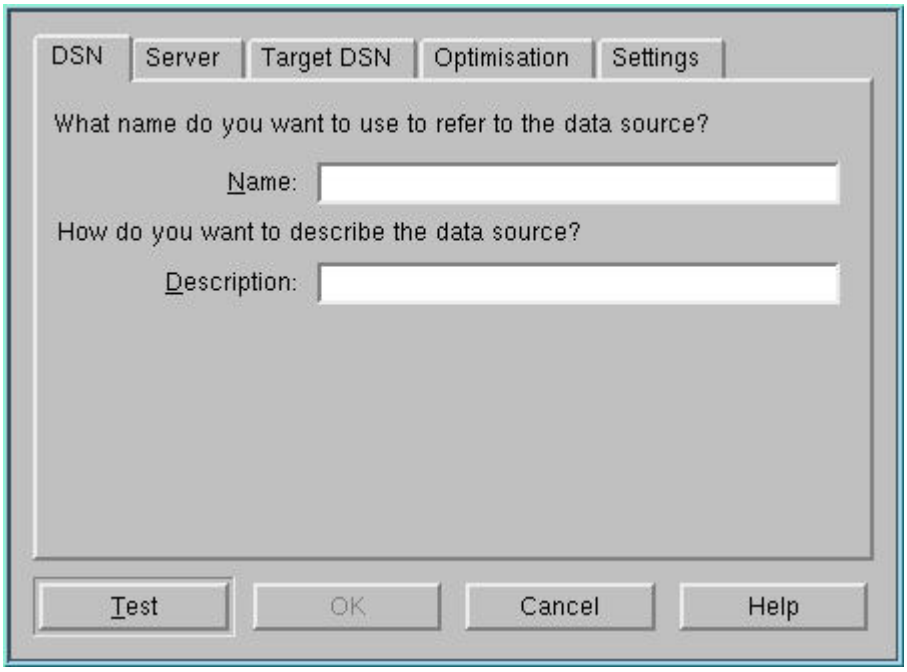
If you're using the GUI ODBC Administrator (ODBCConfig), you can work with data sources by choosing the **User DSN** or **System DSN** tabs:



How do you create an ODBC data source

There are three main ways to create an ODBC data source:

- If your driver has a setup library (examine your `odbcinst.ini` file) then you may be able to define a System or User data source using the unixODBC ODBC Administrator. Start the ODBC Administrator using ODBCConfig, choose **User DSN** or **System DSN**, and then choose **Add**. Choose the ODBC driver you want, then choose **OK**. You should be presented with the ODBC driver's configuration dialog box. Complete the fields and choose **OK**. For example, with the Easysoft ODBC-ODBC Bridge driver you get a tabbed dialog box:



- Edit the System or User DSN INI file (`odbc.ini` or `.odbc.ini`) and add a data source using the syntax:

```
[ODBC_DSN}
Driver = driver_name
Description = description_of_data_source
attributen = value
.
.
attributen = value
```

where, *ODBC_DSN* is this data source's name, *driver_name* is the ODBC driver (refer to `odbcinst.ini` for the available ODBC drivers) and *attributen = value* is the name of an ODBC driver attribute and its value. For example, for the Easysoft ODBC-ODBC Bridge you might define:

```
[my_datasource]
Driver = OOB
Description = Easysoft ODBC-ODBC Bridge DSN
ServerPort = myoobserver:8888
TargetDSN = mytargetdsn
LogonUser = server_user_name
LogonAuth = password_for_LogonUser
```

Check your ODBC driver documentation to find out what attributes you need to define. At a minimum, you must specify the `Driver` attribute and it's always advisable to include the `Description` attribute.

- Create a FileDSN. ODBCConfig does not yet handle File DSNs properly, but you can still use them if they're manually created or produced using the `SAVEFILE` connection attribute to `SQLDriverConnect`. A File DSN definition is basically the same as a DSN definition in the User or System INI files, except that it contains only one data source and the data source is always named `ODBC`. For example:

  ```
  [ODBC]
  Driver = OOB
  Description = Easysoft ODBC-ODBC Bridge DSN
  ServerPort = myoobserver:8888
  TargetDSN = mytargetdsn
  LogonUser = server_user_name
  LogonAuth = password_for_LogonUser
  ```

  Note that File DSNs may be stored anywhere as they are referenced by including `FileDSN=`*path_to_file_dsn* in the connection string.

You can list User and System data sources with:

```
$ /usr/local/easysoft/unixODBC/bin/odbcinst  -q -s
[sqlserver]
```

```
[ODBCNINETWO]
[aix]
[bugs]
[ib7]
[ODBC_JDBC_SAMPLE]
[postgres]
[EASYSOFT_JOINENGINE1]
[SYBASEA]
```

Download ODBC Drivers for Oracle, SQL Server, Salesforce, MongoDB, Access, Derby, InterBase & DB2.

How do you install an ODBC driver?

There are three methods of installing an ODBC driver under unixODBC:

- You write a program that links with `libodbcinst.so` and calls `SQLInstallDriver`.
- You create an ODBC driver template file and run `odbcinst`. For example:

  ```
  odbcinst -f template_file -d -i
  ```

  In this case, your template file must contain the `Driver` and `Description` attributes and may contain the `Setup` attribute. For example:

  ```
  [DRIVER_NAME]
  Description = description of the ODBC driver
  Driver = path_to_odbc_driver_shared_object
  Setup = path_to_driver_setup_shared_object
  ```

- You edit your `odbcinst.ini` file and add the driver definition.

In `odbcinst.ini`, each driver definition begins with the ODBC driver name in square brackets. The ODBC driver name is followed by `Driver` and `Setup` attributes. `Driver` specifies the path to the ODBC driver shared object (which exports the ODBC API). `Setup` specifies the path to the ODBC driver setup library (which exports the `ConfigDriver` and `ConfigDSN` APIs used to: install or remove the driver; create or edit or delete data sources). Few ODBC drivers for Linux or UNIX have a setup dialog box.

To list all installed ODBC drivers, use:

```
$ /usr/local/easysoft/unixODBC/bin/odbcinst  -s -q
[sqlserver]
[ODBCNINETWO]
[aix]
[bugs]
[ib7]
[ODBC_JDBC_SAMPLE]
[postgres]
```

```
[EASYSOFT_JOINENGINE1]
[SYBASEA]
```

What are System and User data sources

System data sources are available to anyone on the machine where the data source is defined. Typically, these are defined in some system defined location that everyone has read access to. For example, `/etc/odbc.ini`. User data sources are defined in a user's home directory in the `.odbc.ini` file. They are only readable by that user (dependent on the value of your umask at the time the file is created).

Whether you can access User DSNs depends on the ODBC driver you are using and whether it is built with unixODBC support.

How your driver locates System and User DSNs depends on whether it was built to use `SQLGetPrivateProfileString` in unixODBC or not. Drivers that know about the unixODBC Driver Manager use the ODBC API `SQLGetPrivateProfileString` to obtain DSN attributes. If a driver does this, it doesn't matter where System or User DSNs are defined, as unixODBC knows where to look for them and what the format of the `odbc.ini` (or `.odbc.ini`) file is. If your driver does not have built in support for `SQLGetPrivateProfileString` then:

1. It will not know where your ODBC data sources are defined.
2. It may not be capable of parsing the `odbc.ini` file format.

ODBC drivers supporting the unixODBC Driver Manager link against `libodbcinst.so` and include `odbcinstext.h`. If you're an ODBC driver writer, we strongly recommend that you install unixODBC and build your ODBC driver with:

```
-I /path/include \
  -L/path/lib -l odbcinst
```

and include `odbcinst.h`.

Some server applications that use ODBC don't support user credentials or change to the specified user, as they run in the context that the server application was started in. In this case, they can't access User DSNs since they are not running as the user the User DSN is defined for. A common error with Apache is to define a User DSN for, say, user Fred and then run Apache under the nobody account. Bridges like the Easysoft ODBC-ODBC Bridge log in as particular user and hence have access to that user's DSNs. If you're using an application that runs as a specific user and you want to use User DSNs, you need to define the User DSN in that user's account (or use a System DSN instead).

Where are ODBC data sources defined?

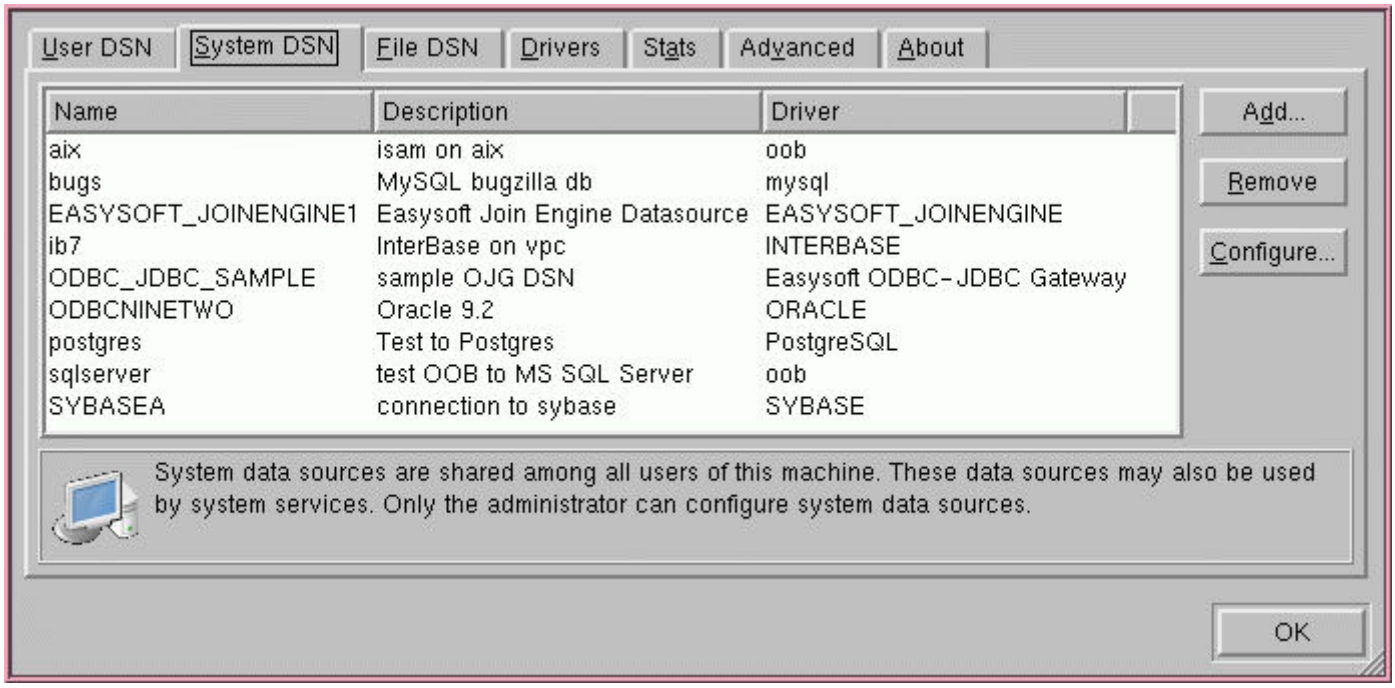
ODBC data sources are defined in two different files depending on whether they are a User DSN or a System DSN. User DSNs are defined in the `.odbc.ini` file in the current user's home directory. The System DSN location is defined with `--sysconfdir` at compile time for unixODBC. You can locate this directory with:

```
$ odbcinst -j
unixODBC 2.3.1
DRIVERS............: /etc/odbcinst.ini
SYSTEM DATA SOURCES: /etc/odbc.ini
FILE DATA SOURCES..: /etc/ODBCDataSources
USER DATA SOURCES..: /home/auser/.odbc.ini
SQLULEN Size.......: 4
SQLLEN Size........: 4
SQLSETPOSIROW Size.: 2
```

In this case, User DSNs are defined in `/home/auser/.odbc.ini` because the user running the `odbcinst` command was auser and auser's home account is `/home/auser`.

You can tell unixODBC to look in a different file for System DSNs by defining and exporting the `ODBCINI` environment variable. Include the file name and path when setting this variable.

If you're using the GUI ODBC Administrator (ODBCConfig), you can access data sources from the **User DSN** and **System DSN** tabs:



What does a data source look like?

Generally speaking, a DSN is comprised of a name and a list of attribute-value pairs. Usually, these attributes are passed to the ODBC API `SQLDriverConnect` as a semicolon delimited string. For example:

```
DSN=mydsn;attribute1=value;attribute2=value;attributen=value;
```

What a specific ODBC driver needs is dependent on that ODBC driver. Each ODBC driver supports a number of ODBC connection attributes, which are passed to the ODBC API `SQLDriverConnect`. Any attributes that aren't defined may be looked up in the DSN defined in the ODBC connection string. For

example, suppose your ODBC application calls `SQLDriverConnect` with the connection string `DSN=mydsn;` but it needs the name of the server where the database is located. Since the connection string does not contain the attribute, this driver needs to locate the server (for example, `Server=xxxxx`). The ODBC driver can look up the DSN `DSN=mydsn;` and check whether this defines a `Server` attribute.

Any driver that supports unixODBC uses `SQLGetPrivateProfileString` to lookup any attributes it needs using the DSN name as a key. Normally, your ODBC application either passes all the attribute=value pairs in the connection string or lets you choose a DSN from a list then calls `SQLDriverConnect("DSN=mydsn;")`. In the latter case, the ODBC driver looks up the additional attributes in the DSN definition.

Each ODBC driver defines the attributes it needs to connect to a particular database. For example each Easysoft ODBC-0DBC Bridge DSN must define `TargetDSN`, `LogonUser`, `LogonAuth`, and `ServerPort`.

For unixODBC, System DSNs are defined in an `odbc.ini` file in the system defined path and User DSNs are defined the current user's home directory (in a file called `.odbc.ini`). The format of this file is:

```
[DSN_NAME]
Driver = driver_name_defined_in_odbcinst.ini
attribute1 = value
attribute2 = value
.
.
attributen = value
```

Testing DSN connections

Once you've installed your ODBC driver and defined an ODBC data source, you can test the connection to it by using unixODBC's isql utility. For example:

```
isql -v DSN_NAME db_user_name db_password
```

You should use the `-v` option because this causes isql to output any ODBC diagnostics if the connection fails. *db_user_name* and *db_password* are optional, but you must supply them if your ODBC driver requires a database user name and password to log into the DBMS.

If isql successfully connects to your DSN, it should display a banner and a `SQL>` prompt:

```
bash-2.05$ isql -v my_dsn my_user my_password
+---------------------------------------+
| Connected!                            |
|                                       |
| sql-statement                         |
| help [tablename]                      |
| quit                                  |
```

```
|                                       |
+---------------------------------------+
SQL>
```

If the connection fails (and you specified `-v`) then any ODBC diagnostic from the ODBC driver explaining why it could not connect should be displayed.

```
$isql -v mysql_db my_user my_password
[unixODBC][MySQL][ODBC 3.51 Driver]
Access denied for user 'my_user'@'xxx.easysoft.local' (using password: YES)
[ISQL]ERROR: Could not SQLConnect
```

What this ODBC diagnostic says depends on the ODBC driver and you should look up it in the documentation for your ODBC driver.

Some errors may be reported by the unixODBC Driver Manager itself (if for instance it could not connect to the ODBC driver). An example is:

```
$isql -v dsn_does_not_exist my_user my_password
[unixODBC][Driver Manager]
Data source name not found, and no default driver specified
[ISQL]ERROR: Could not SQLConnect
```

In this case, unixODBC couldn't locate the DSN `dsn_does_not_exist` and therefore couldn't load the ODBC driver. Common reasons for this error are:

- The DSN `dsn_does_not_exist` doesn't exist in your User or System INI files.
- The DSN `dsn_does_not_exist` does exist in an INI file, but you have omitted the `Driver=xxx` attribute telling the unixODBC Driver Manager which ODBC driver to load.
- The `Driver=`*`path_to_driver`* in the `odbcinst.ini` file points to an invalid path, to a path to an executable where part of the path is not readable/searchable or to a file that is not loadable (executable).
- The `Driver=xxx` entry points to a shared object that doesn't export the necessary ODBC API functions. (You can test this with dltest, which is included with unixODBC.)
- The ODBC driver defined by `DRIVER=xxx` in the `odbcinst.ini` file depends on other shared objects that aren't on your dynamic linker search path. Run `ldd` on the driver shared object (named by `Driver=` in the `odbcinst.ini` file) to find out what dependent shared objects can't be found. If some can't be found, set your `LD_LIBRARY_PATH` environment variable to specify these dependent shared objects paths or add these paths to `/etc/ld.so.conf` and rerun `ldconfig`.

Further information

- Why do I get error "Data source name not found and no default driver specified"?

isql beyond testing connections

Note Unless you're running isql in batch mode, we strongly suggest include the `-v` (verbose) argument because that retrieves ODBC diagnostics on failed commands and other useful information. The examples in this section assume isql was run with the `-v` argument.

Although isql can be used to test the connection to your data sources, it can do quite a lot more. Once connected to your data source, you're provided with an SQL prompt at which you can:

- Enter SQL, which is sent to the ODBC driver you're connected to.
- Obtain the result set from an `SQLTables` call to return a list of tables in your database. Just enter `help` at the prompt.
- Obtain the result set from an `SQLColumns` call to return a list of column definitions in a table. Just enter `help` *`tablename`*.

isql passes what you enter at the SQL prompt to the ODBC driver (assuming what you enter is not an isql command). isql uses `SQLPrepare` then `SQLExecute`. If the `SQLExecute` call fails (or returns `SQL_SUCCESS_WITH_INFO`), isql uses `SQLError` to obtain ODBC diagnostics. For example:

```
SQL> select * from table_does_not_exist
[S0002][unixODBC][Microsoft][ODBC SQL Server Driver][SQL Server]
Invalid object name 'table_does_not_exist'.
[37000][unixODBC][Microsoft][ODBC SQL Server Driver][SQL Server]
Statement(s) could not be prepared.
[ISQL]ERROR: Could not SQLExecute
SQL>
```

If the `SQLExecute` call for your SQL succeeds, isql uses `SQLNumResultCols` to ascertain if the SQL returned a result set. If a result set is found, isql fetches it and displays the result set. Format the result set by using the command line settings `-d` or `-x` (how to delimit columns), `-w` (output in HTML table), `-c` (column names on first row if `-d` or `-x` are used) and `-m` (limit column display width).

After any SQL succeeds, isql calls `SQLRowCount` to see how many rows were affected. You should note that many ODBC drivers return `-1` if the SQL was a result set generating statement, otherwise this should be the number of rows inserted, deleted, or updated.

As each command or SQL statement entered at the prompt and terminated with a newline is passed to the ODBC driver, you can run isql with stdin redirected to a file containing SQL. For example, suppose you create the file `myfile.sql` containing:

```
create table test (a integer)
insert into test values (1)
insert into test values (2)
```

you can then use:

```
isql -v mydsn dbuser dbauth < myfile.sql
```

to execute multiple SQL commands in one go. Obviously, you can also redirect stdout.

Tracing ODBC calls

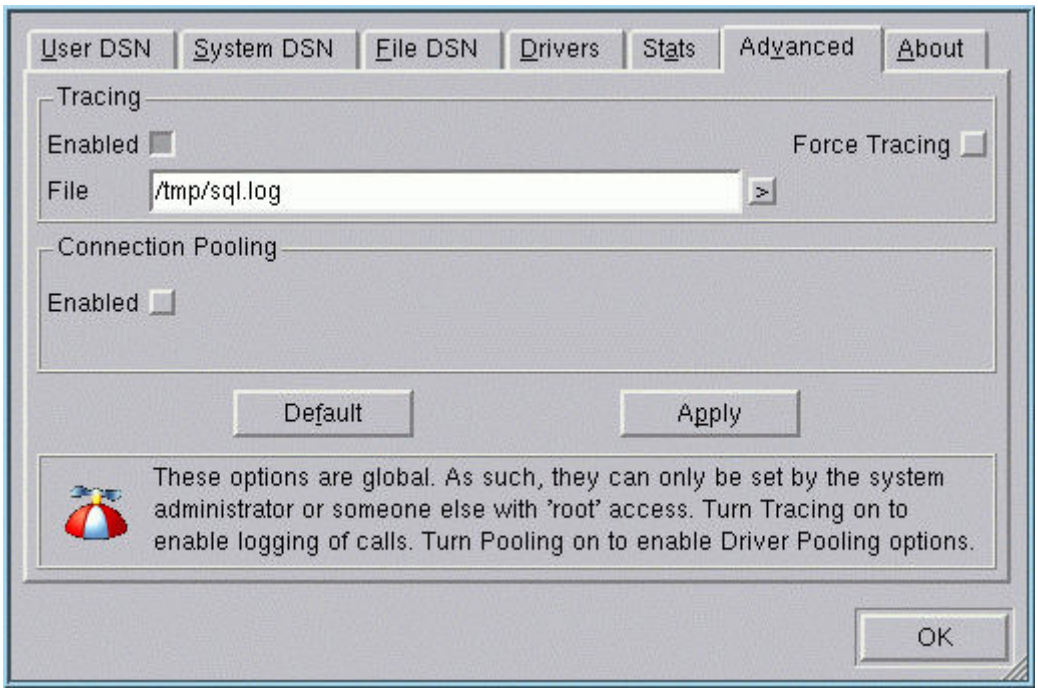
The unixODBC Driver Manager can write a trace of all ODBC calls made to a file. This can be a very useful debugging aid but it should be remembered that tracing will slow your application down. You enable tracing using one of the following methods:

- Locate your `odbcinst.ini` file and add a section to this file like:

  ```
  [ODBC]
  TraceFile = /tmp/sql.log
  Trace = Yes
  ```

  You can use any file for the `TraceFile` argument and it does not need to pre-exist. The permissions on the `odbcinst.ini` may be such that you need to be the root user.

- You can enable tracing and define the trace file using the ODBC Administrator (ODBCConfig).



Be careful when running ODBC applications as different users because most users set their umask such that other users cannot write to newly created files. If user A enables tracing and connects to the Driver Manager, the trace file will be created. When user B uses the Driver Manager, it's likely nothing gets traced because user B does not have write permission to the trace file.

Trace files generally contain a log of each entry and exit to each ODBC API. For example:

```
[ODBC][9155][SQLAllocHandle.c][345]
                Entry:
                        Handle Type = 2
                        Input Handle = 0x80899d0
[ODBC][9155][SQLAllocHandle.c][463]
                Exit:[SQL_SUCCESS]
                        Output Handle = 0x8089f60
```

The general form is:

```
[ODBC][Process ID][C source containing the ODBC API][source line number]
      Entry:
         argument 1 = value
         argument 2 = value
         argument n = value
[ODBC][Process ID][C source containing the ODBC API][source line number]
      Exit: [ODBC status]
            output argument 1 = value
            output argument 2 = value
            output argument n = value
```

With this tracing you can find out:

- Every ODBC API that was called and in what order.
- The arguments provided to each ODBC API.
- Any values returned by an ODBC API
- The exist status of each ODBC API

If a serious error occurs, which could be a problem in unixODBC, you can find out the line number in the unixODBC source file where the error was generated.

What does the cursor library do?

The cursor library is included in unixODBC for applications that require cursor types the ODBC driver does not support. Whether the cursor library is used depends on:

- How the application calls `SQLSetConnectAttr` for the attribute `SQL_ODBC_CURSORS`. The default (if `SQLSetConnectAttr` is not called to set `SQL_ODBC_CURSORS`) is `SQL_CUR_USE_DRIVER`, which means use cursors in the ODBC driver. (If you need cursors and the driver does not support the one you require, the application will fail.) Other values for `SQL_ODBC_CURSORS` are `SQL_CUR_USE_IF_NEEDED` (which means the cursor library will be used if you attempt to use a cursor the driver does not support) and `SQL_CUR_USED_ODBC` (which means to use the cursor library in unixODBC).
- Whether you bind result set values. The cursor library does not work for fetching results with `SQLGetData`. It only works if you issue a query, bind the columns to variables with `SQLBindCol`, and then call `SQLFetch`.

The cursor library is a shared object called `libodbccr.so`, which is in the `lib` subdirectory of wherever you set `--prefix` to when you build unixODBC. When the cursor library is in use, the normal ODBC entry points to the ODBC driver are replaced with entry points in the cursor library, which then go on to call the same entry points in the ODBC driver, but apply extra processing to imitate the required cursor.

Setting ODBC driver environment variables automatically

- `DMEnvAttr` and `SQL_ATTR_UNIXODBC_ENVATTR`

This is a data source setting specified in the `odbc.ini` file. This sets ODBC environment attributes. The form is:

```
DMEnvAttr = ATTRIBUTE_NAME=value
```

or, if *value* contains spaces:

```
DMEnvAttr = ATTRIBUTE_NAME={value}
```

where *ATTRIBUTE_NAME* is the name of an ODBC environment attribute. (For example, `SQL_ATTR_CONNECTION_POOLING`.)

unixODBC defines a new environment attribute for itself called `SQL_ATTR_UNIXODBC_ENVATTR`. If your driver needs some environment variables to be set (for example, `ORACLE_HOME` or `DB2INSTANCE`) you can set them with `SQL_ATTR_UNIXODBC_ENVATTR`:

```
DMEnvAttr = SQL_ATTR_UNIXODBC_ENVATTR={envvar=value;envar=value}
```

For example:

```
DMEnvAttr = SQL_ATTR_UNIXODBC_ENVATTR= {ORACLE_HOME=/opt/OraHome}
```

sets the `ORACLE_HOME` environment variable to `/opt/OraHome` before loading the Oracle ODBC driver.

- `DMConnAttr` and `DMStmtAttr`

  These unixODBC-specific data source attributes work like `DMEnvAttr`. The format is:

  ```
  DMConnAttr = CONNECTION_ATTRIBUTE=value
  DMStmtAttr = STATEMENT_ATTRIBUTE=value
  ```

  where:

  - *CONNECTION_ATTRIBUTE* is the name of an ODBC connection attribute (for example, `SQL_ATTR_CONNECTION_TIMEOUT`).
  - *STATEMENT_ATTRIBUTE* is the name of an ODBC statement attribute (for example, `SQL_ATTR_NOSCAN`).
  - *value* is the value you want to set the attribute to. For example, `SQL_ATTR_CONNECTION_TIMEOUT=30 or SQL_ATTR_NOSCAN=SQL_NOSCAN_OFF.`

  `For example:`

  ```
  DMConnAttr = SQL_ATTR_AUTOCOMMIT=SQL_AUTOCOMMIT_OFF
  ```

  Note `If you prefix the attribute name with a *, this fixes the value of that attribute. Any attempt by the application to set that attribute value will be ignored and unixODBC will replace the value with that specified in the DMxxxAttr.`

Unicode in unixODBC

The original ODBC specification was mostly written by Microsoft and handed over to X/Open. However, since then Microsoft have made a number of changes to their ODBC specification, including adding some support for Unicode.

Unicode in ODBC is supported by the so-called wide APIs (because every supported ODBC API has an equivalent one ending in W) and some new types like SQL_WCHAR. The wide APIs accept and return UCS-2 encoded data. There is also a macro definition in the C headers (UNICODE) that determines if any call to the ODBC API SQLxxx ends up calling SQLxxxA (ANSI version) or SQLxxxW (wide version). However, don't define this unless you're sure all the data you will pass to the ODBC APIs is really UCS-2.

Supported Unicode APIs

There is an equivalent wide API for almost every ANSI ODBC API. For example, SQLPrepareA expects 8-bit characters and SQLPrepareW expects UCS-2. A notable omission is SQLGetData. Because SQLGetData accepts a type you want the data returned as (you can ask for SQL_CHAR or SQL_WCHAR). You can find a list of wide APIs in sqlucode.h, which is included with your ODBC Driver Manager.

Mismatched applications and drivers

Not all ODBC drivers support Unicode and not all applications support Unicode, so the unixODBC Driver Manager has some work to do when there is a mismatch. In general, ODBC Driver Managers always call the wide APIs if the ODBC driver supports them even if the application is ANSI. (For example, the Microsoft ODBC Driver Manager does this.) The ODBC Driver Manager needs to convert characters to UCS-2 first. A similar issue arises with Unicode data returned by the ODBC driver to an ANSI application, only in this case, data is effectively lost if it doesn't fit into 8 bits. However, unixODBC attempts to sidestep that work and if it spots the application is ANSI, it uses the ANSI ODBC APIs in the ODBC driver. If you truly want to always use the wide APIs in a supporting ODBC driver (because your application supports Unicode), you must tell unixODBC by calling SQLDriverConnectW.

Note unixODBC does not get involved with returned bound column data or sent bound parameters. If you bind an SQL_WCHAR, it should be returned as wide characters and you should set parameters as wide characters.

But UCS-2 is not Unicode

Correct, UCS-2 is an encoding that supports up to 64K characters (up to 0xFFFF) and Unicode contains a lot more than that. More recent versions of some ODBC drivers and databases support UTF-16 and hence surrogate pairs.

For example, Microsoft SQL Server 2012 introduces a new collation sequence suffix (_SC) and it supports surrogate pairs.

However, as always, be careful when using SQLGetData.

But Unicode on UNIX is stored in wchar_t

wchar_t can be 2 bytes or 4 bytes on various Linux and UNIX platforms and is totally incompatible with ODBC, which uses UCS-2 and UTF-16.

UTF-8 and ODBC

Some ODBC drivers support the sending and receiving of character data encoded in UTF-8 and therefore all of the Unicode character set can be supported. However, there is potential problem with SQLGetData. Usually, a driver has some flag to enable this and you continue to use the ANSI APIs and not the wide APIs.

For example, the Easysoft SQL Server ODBC driver has a flag called ConvToUTF. When enabled, UTF-8 encoded data sent to Microsoft SQL Server is converted to UCS-2 and returned data is converted from UCS-2 to UTF-8. This flag is enabled in the following example, in which isql, an ANSI application that uses the ANSI APIs, retrieves some Unicode data from a SQL Server database:

```
$ /usr/local/easysoft/unixODBC/bin/isql.sh -v MY_SQL_SERVER_DSN
SQL> select ncharcol from my_table
+------------+
| ncharcol   |
+------------+
| Ű́ñĩćōđě    |
+------------+
```

The SQLGetData problem

When you call SQLGetData, you specify the type you want the data returned as and a buffer to accept the data. Obviously, if you're asking for Unicode data. you'd better make sure that your buffer length (in bytes) is divisible by 2.

Some of the ODBC APIs are declared in terms of characters and some in terms of bytes and some we're not sure about (for example, SQLGetData).

SQLGetData returns the length or indicator value in StrLen_or_IndPtr, which is not defined as bytes or characters so, if too small, a buffer is passed to SQLGetData for the column. In this case, does the StrLen_or_IndPtr contain the number of characters required in the buffer to retrieve the whole column or the number of bytes? Some applications call SQLGetData with a zero-length buffer simply to find our how big a buffer to pass for real by looking at StrLen_or_IndPtr.

Also, the ODBC specification says if you call SQLGetData with a buffer that is too small, it will fill the buffer and you need to call SQLGetData again to get the remaining data.

So what if you're using a driver that supposedly does UTF-8 and you pass SQLGetData a buffer of *n* bytes, but *n*+1 bytes were required and the last character in the buffer requires 2 bytes in UTF-8 encoding? Does the driver part fill the buffer leaving one whole UTF-8 character off or does it fill the buffer thus leaving you with half a character? If it does not fill the buffer then it contradicts the ODBC specification and if it does you cannot use your data until you have retrieved all of it.

Drivers supporting UTF-8 seem to handle this in different ways, but the only safe way for an application to deal with it is to ensure you pass a big enough buffer in the first place (in which case you might as well bind the column).

Incidentally, the same issue exists with UTF-16 and asking for SQL_WCHAR characters.

ODBC 3.8 Support in unixODBC

To get the most complete support for ODBC 3.8, you currently need to check out and build the unixODBC source code. For example:

```
$ svn co svn://svn.code.sf.net/p/unixodbc/code/trunk unixodbc-code
$ cd unixodbc-code
$ make -f Makefile.svn
$ configure
$ make
# make install
```

This version of unixODBC includes the header files and API changes required for ODBC drivers to support driver-aware connection pooling, asynchronous connection operations, and streamed output parameters.

Other unixODBC Utilities

The unixODBC distribution includes a few binaries, which can prove useful:

odbcinst

The odbcinst binary can be used to perform a number of unixODBC administration functions or query the unixODBC configuration. If you run odbcinst without any arguments, it lists all the things you can do with it. Here are a few examples:

odbcinst -j prints the unixODBC configuration which includes:

- Where your driver INI file is.
- Where your default User and System DSNs are defined.
- The sizes of some ODBC types (for information on this, please refer to 64-bit ODBC).

odbcinst -q -d lists all the ODBC drivers that have registered with unixODBC.

odbcinst -q -s lists all the ODBC data sources you have defined.

There are other options to install DSNs and ODBC drivers.

odbc_config

This utility is only in more recent unixODBC versions.

Use odbc_config to find out how unixODBC was built and also some of the information odbcinst provides. This is usually most useful for people building applications and drivers against unixODBC. For example, Perl's DBD::ODBC module can use it to find the CFLAGS required to build its ODBC XS code. Some examples are:

```
$ odbc_config --odbcini # show the System DSN ini file
   /etc/odbc.ini
$ odbc_config --odbcinstini # show the driver ini file
  /etc/odbcinst.ini
$ odbc_config --prefix # shows what --prefix was set to when configured
  /usr/
$ odbc_config --libs # lib line to add to linker
  -L//usr/lib -lodbc
```

Appendix A: unixODBC INI files format

unixODBC uses three INI files:

- odbcinst.ini defines installed or registered ODBC drivers and where to find them.
- odbc.ini defines ODBC data sources.
- *any_file* file DSNs accessed by using FILE=*any_file* in the connection string.

In all these files, the following conventions apply:

- A # or ; character at the start of a line means the rest of the line is a comment and are ignored. **Note** # or ; characters anywhere else on a line other than the first characters are interpreted literally.
- Sections of an INI file begin with a string in square brackets [].

  In the odbcinst.ini file, the section defining a driver begins with the driver name within [ ].

  In the odbc.ini file, the DSN name is placed within [ ].

  In a file DSN, the data source name is always ODBC, ([ODBC]) and there can only be one in each file.

- When specifying attributes, for example, attribute_1 = value_1, the white space either side of the assignment operator (=) is ignored but white space elsewhere is taken literally. For example, attribute_1 = attribute 1 value assigns the value attribute 1 value to attribute_1.
- In general, you should avoid using braces {} unless your driver documentation tells you otherwise, as an existing issue with unixODBC stops anything after a line containing {} from being parsed.

Appendix B: unixODBC installed files

If you build unixODBC from the source distribution and restrict the build to unixODBC and not any of the included drivers (--enable-drivers=no), the following files are installed:

- libodbc.so, the ODBC Driver Manager.

  ODBC applications link to this to access ODBC drivers.

- libodbcinst.so, the ODBC Driver Manager library.

  ODBC drivers link with this to access the SQLGetPrivateProfileString and other driver APIs. Installers might also link to this library to use SQLInstallDriver.

- libodbccr.so, the ODBC cursor library
- dltest, a binary that lets you check for the existence of shared object entry points.
- isql, a small example ODBC application that lets you run queries against your ODBC drivers.
- ODBCConfig, a GUI application that lets you install, edit, create, and delete ODBC drivers and data sources.
- odbcinst, a small binary that lets you install, create, and delete ODBC drivers and data sources. It can also return various unixODBC

configuration details. For example, the version.

- Various C header files:

  odbcinstext.h, odbcinst.h, sqlext.h, sql.h, sqltypes.h, and sqlucode.h.

Note Not all UNIX platforms use .so as the shared object file extension. Some versions of HP-UX use .sl and AIX uses archives (.a) containing shared objects using the .o extension.