Dealing with SQLite Connection Errors

Whil Hentzen

"We need to get away from DBFs" is a refrain I hear regularly from fellow developers. Be it due to perceived instability of the file format, the need for tables larger than 2 GB, or the result of political machinations, the result is the same – a desire to move to a SQL database back-end. SQLite can be an excellent intermediate step – and possibly the final word - in the process of restructuring your application to talk to a SQL back-end. In this article, I discuss how to deal with the usual spate of errors that can occur when connecting to SQLite from VFP.

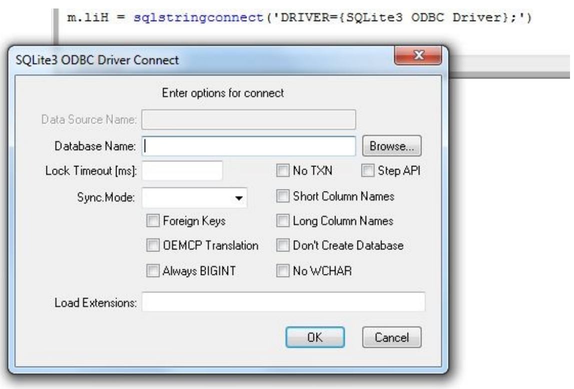
The true hallmark of an amateur, it is said, is the squeal of delight when their attempt at accomplishing a technical task works, because the amateur thinks they're done, and is ready to move on to the next step in the process.

Those of us who have been around the block a few times know that the first success simply means that the proof of concept step has been passed - that the goal is technically "possible". The next step is to lay the foundation for all of the problems that could occur, because, inevitably, they will.

In my last article, I showed how to set up SQLite and connect to it from within VFP. Proof of concept achieved. Let's now look at what could go wrong.

SQLite needs the database name and path

The first error you might run across when trying to connect is if you forget to include the database name in the connection string, or if you use a path that doesn't exist. It's easy to troubleshoot, though, because the result of this error is VFP displaying the SQLite3 ODBC Driver Connect dialog as shown in **Figure 1**.



```
m.liH = sqlstringconnect('DRIVER={SQLite3 ODBC Driver};')
```

SQLite3 ODBC Driver Connect

Enter options for connect

Data Source Name:
Database Name:
Lock Timeout [ms]:
Sync.Mode:
Load Extensions:

Figure 1. Not including a valid database file name and path in the connection string forces the Driver Connect dialog to be displayed.

There are some rules about what works and what doesn't in terms of a valid database name and path. Here's the logic used by the connection code:

1. Is a Database string provided in the Connection String? If no, open the Driver Connect dialog. If yes, go to Step 2.

2. Is a valid path provided in the name and path string? If no, open the Driver Connect dialog, with the string that was provided displayed as the default value in the Database Name textbox. If yes, go to Step 3.

3. Does the database file that was specified in the name and path string exist? If no, create a new file. If yes, connect to the file.

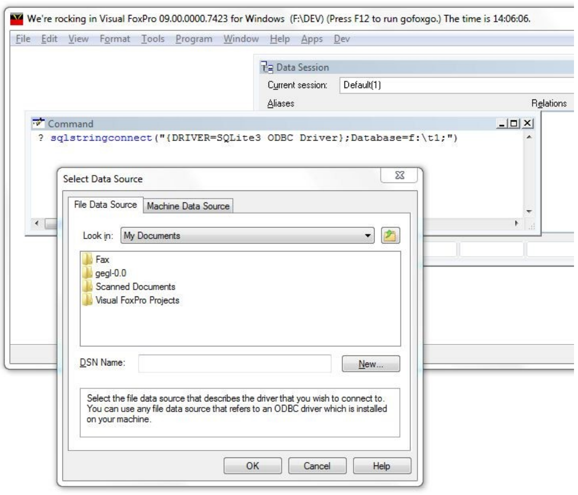
Typographic errors

The next error you will likely run into is a simple typographic error in the connection string.

If you've mistyped the connection string, like so:

```
m.liH = sqlstringconnect("{DRIVER=SQLite3 ODBC
Driver};Database=f:\t1")
```

(the error is that the opening braces is before the word "DRIVER" instead of before the word "SQLite," a common problem), you'll get a dialog box asking you where the driver is, as shown in Figure 2.


We're rocking in Visual FoxPro 09.00.0000.7423 for Windows (F:\DEV) (Press F12 to run gofoxgo.) The time is 14:06:06.

Data Session — Current session: Default(1)

```
? sqlstringconnect("{DRIVER=SQLite3 ODBC Driver};Database=f:\t1;")
```

Select Data Source

Look in: My Documents

Fax
gegl-0.0
Scanned Documents
Visual FoxPro Projects

DSN Name:

Select the file data source that describes the driver that you wish to connect to. You can use any file data source that refers to an ODBC driver which is installed on your machine.

Figure 2. The Select Data Source dialog appears if you mistype a driver name.

Easy enough. But what about those not as easily troubleshot? The next type of trouble will be indicated when you don't get an immediate response; typically, a successful connection returns a value before you can blink. A connection attempt that fails can take 5 to 10 seconds before the dreaded '-1' appears.

For example, using parens instead of braces to surround the driver name will cause the SQLStringConnect command to simply return a -1.

Bad:

```
m.liH = sqlstringconnect("(DRIVER=SQLite3 ODBC
Driver);Database=f:\t1")
```

Good:

```
m.liH = sqlstringconnect("{DRIVER=SQLite3 ODBC
Driver};Database=f:\t1")
```

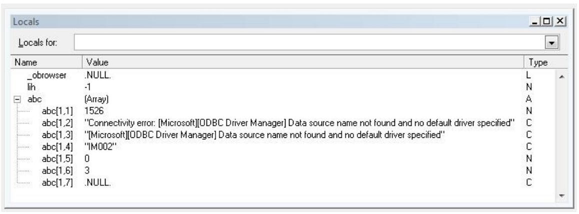
Not very obvious, but eventually detectable. Other times, errors can be even more obscure. Here's how to gather better information about what's happening.

Gathering error information

Error information on the ODBC connection attempt can be captured via the AERROR function, like so:

```
m.lcXN = "{DRIVER=SQLiteXXX ODBC
Driver};Database=f:\t1"
liHandle = sqlstringconnect(lcXN)
dimension abc[1]
? aerror(abc)
```

In this example, you may have already noticed that the name of the driver is invalid – no such thing as SQLiteXXX as of this writing (or probably ever) - thus generating an error. Open up your Locals window (found under the Tools menu if the Debugger is in the FoxPro frame or under the VFP Debugger's Window menu if you're running the debugger in its own frame) as shown in Figure 3.


Locals

Locals for:

Name	Value	Type
_obrowser	.NULL.	L
lih	-1	N
abc	(Array)	A
abc[1,1]	1526	N
abc[1,2]	"Connectivity error: [Microsoft][ODBC Driver Manager] Data source name not found and no default driver specified"	C
abc[1,3]	"[Microsoft][ODBC Driver Manager] Data source name not found and no default driver specified"	C
abc[1,4]	"IM002"	C
abc[1,5]	0	N
abc[1,6]	3	N
abc[1,7]	.NULL.	C

Figure 3. The Locals window allows you to easily drill down into an AERROR array.

Spelunk through the results. Unfortunately, the errors returned by the driver can sometimes be pretty misleading. In this example, I specified a bad driver name, which means VFP and ODBC weren't going to find "SQLiteXXX ODBC Driver" in the Windows registry. I don't think that "Data source name not found" doesn't make sense – given that I didn't specify one at all – and "no default driver specified" was similarly obscure. The error should have been something like "Could not find driver."

If you don't want to go through the trouble of using the Locals window, you can simply issue a

```
display memory like abc
```

command and view the results on the VFP desktop.

```
abc[1,1] = 1526
abc[1,2] = "Connectivity error: [Microsoft]
[ODBC Driver Manager] Data ..."
abc[1,3] = "[Microsoft][ODBC Driver Manager]
Data source name..."
abc[1,4] = "IM002"
abc[1,5] = 0
abc[1,6] = 3
abc[1,7] = .NULL.
```

An alternative to AERROR()

It can be a nuisance to have to go through the DIMENSION and AERROR rigamarole each time you run into an error during development. You can get VFP to throw a dialog displaying errors via the SqlSetProp() function, like so:

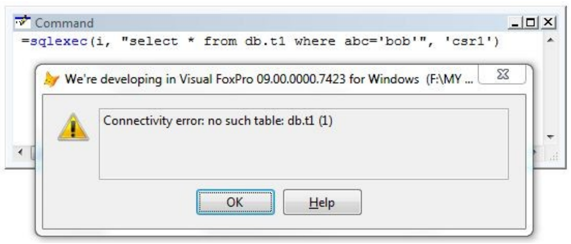
```
=SQLSetProp(liH,"DispWarnings",.t.)
```

The error description will be shown in a dialog as shown in Figure 4.

You need to use the DIMENSION and AERROR commands in a production system so as to protect the user from unfriendly error messages that they don't know how to deal with, so you'll definitely want to turn this setting OFF in your applications by issuing a

```
=SQLSetProp(liH, "DispWarnings", .f.)
```

command in your application's setup.



```
=sqlexec(i, "select * from db.t1 where abc='bob'", 'csr1')
```

We're developing in Visual FoxPro 09.00.0000.7423 for Windows (F:\MY ...

Connectivity error: no such table: db.t1 (1)

Figure 4. After setting DispWarnings to True via SQLSetProp(), errors will be displayed in a dialog.

DSNs or connection strings?

As with everything in Visual FoxPro, there are a dozen ways to perform a task. Sometimes even thirteen ways. And if you ask 13 developers which way is the best, each will argue vociferously that their way is the best. So how do you choose? Specifically, what are the pros and cons of DSNs and connection strings?

The primary advantage of a DSN is that it is stored in the Windows Registry, which keeps it reasonably secure from prying eyes and others who would inadvertently or malevolently cause problems. (You could even prevent access to it altogether using Windows' Group Policies, a topic worth investigating but way beyond the scope of this article.) Furthermore, it is possible to create a routine to check for the DSN upon application startup. Chapter 13 of MegaFox, "How can I be sure users have the correct settings," has details.

You could even reinstall the DSN programmatically during startup, exporting the ODBC settings from the Windows Registry as a .REG file, and then execute it on other machines using ShellExecute(), if you needed to. This way users can update settings automatically on application startup and DBAs can retain control over connection information.

The main disadvantage of DSNs, especially System DSNs, is that they are available to any application running on your system. A fairly smart user then could fire up a copy of, say, Microsoft Excel and connect to your data using the DSN which has the username/password stored in it.

The advantage of connection strings is that they are infinitely flexible, as they are built at run-time. It's not a difficult task to store connection string parameters in the system's meta data (encrypted, please, if your connection string includes user logins and passwords) and retrieve them as necessary. Some developers may feel more comfortable manipulating system meta data instead of having to get into the Windows Registry.

I personally prefer connection strings to DSNs, but that's just a personal preference. I don't have a passionate view of one over the other, and suggest that you try on both for size, and see which fits to your liking.

Next issue, we'll start manipulating databases.

Author Profile

Whil Hentzen is a independent software developer based in Milwaukee, Wisconsin (as opposed to Milwaukee, Minnesota, as many people think.) His writing has killed many trees over the years, but none since 2007. He has realized he really sort of misses it. You can reach him at whil@whilhentzen.com