DBTool User Manual 2.4

Qiang He (Ph.D.)

obase@tom.com
qhe@tsinghua.org.cn

http://energy.51.net/dbtool/index.htm

Contents

List of Figures

Chapter 1

Introduction

1.1 What's DBTool

Besides Database Toolbox released by Mathworks, DBTool is another choice to access database from MATLAB.

1.2 DBTool features

The kernel of DBTool is a mex file `dbtool.dll`, which is written and compiled in Visual C++ using the MFC classes CDatabase, CRecordset and some direct ODBC calling. It's reliable and runs faster than Database Toolbox, which is implemented in Java. And then a set of '.m' files is written to wrap it into two MATLAB class objects: `dbase` and `rset`.

A set of functions which make direct calling into `dbtool.dll` is also implemented. They are functional equivalent to there class correspondences. They use structures instead of class objects. This set of functions is named `Plain API`. With the Plain API, you can write MATLAB Compiler (and also MATLAB COM Builder) compatible program, so you can compile your database program into standalone executable. MATLAB Compiler can't compile Database Toolbox program, because it uses class and Java.

MATLAB Database Toolbox does not support binary large object (BLOB) fields, while DBTool can read and write BLOB fields freely. For BLOB fields, the contents are treated as byte streams. All the bytes are read into a MATLAB 1*N double array, each element stores a byte. And any 1*N double array (elements must be 0∼255 integer) can be written into a BLOB field.

DBTool has the following features:

- Faster and reliable, easy to use
- Implemented using MATLAB class objects, also Plain API provided
- MATLAB Compiler and COM Builder compatible with the Plain API
- Designed to access any database which has an ODBC interface
- Designed to access any data types including binary large object(BLOB)
- Multi-rows fetching in one statement
- Directly execute SQL statements
- Automatic close all database when MATLAB is closed
- Enumerate DSN names and table names
- many others...

1.3 Order DBTool

DBTool is shareware. The demo version of DBTool is free for use, but has some limitations:

- Multi-rows fetch not enabled.
- You can open only 1 database and 1 recordset each time.
- The length of BLOB fields is limited to 8192 bytes.

Since version 2.4, the demo version is 30 minutes full functional each time. That is, every time you start DBTool demo version in MATLAB, it acts as professional version for 30 minutes, and then go back to demo version. `dbase` and `rset` objects opened will be stiil alive untill closed.

Besides demo version, the standard, professional and redistributable versions are also available. The demo, standard and professional versions are for personal use only. For commercial use, please buy the redistributable version. Features and prices for different versions are listed in the table below.

Version	Demo	Standard	Professional	Redistributable
BLOB Size	8K	256K	Unlimited	Unlimited
dbase objects	1	16	16	16
rset objects	1	16	16	16
multi-rows fetch	no	no	yes	yes
Plain API	yes	yes	yes	yes
price	free	USD30	USD60	USD300

If you have a PayPal account, please go to the DBTool registration page:

http://energy.51.net/dbtool/purchase.htm

and click the PayPal icon in the price table to register. If you do not have a PayPal account, you can also click the icon, you will be guided to setup a new PayPal account for free.

If you can't pay with PayPal, please send the registration fee to:

```
BANK OF CHINA, BEIJING BRANCH
NO.8 YA BAO LU
BEIJING, CHINA
SWIFT CODE: BKCHCNBJ110

Name    : Qiang He
Account: 4080603-0188-017731-7
```

And send me an email with:

- Your name (or names, each name for a copy)
- Your company
- Your email
- License type and number of copies

Note: If you want to buy several copies of DBTool, different registration information should be supplied for each copy.

Wire transfer is preferred. If wire transfer is not convenient for you, a check via ordinary mail is also acceptable, in this case please contact me to ask for my post address. I'll email the license file to you immediately.

The author's email is: `obase@tom.tom` or `qhe@tsinghua.org.cn`

Chapter 2

Install DBTool

2.1 Demo version installation

2.1.1 Unpack dbtool.zip

Unpack all the files to a folder using WinZIP, such as `"c:\matlab\toolbox\dbtool"`. See figure 2.1.

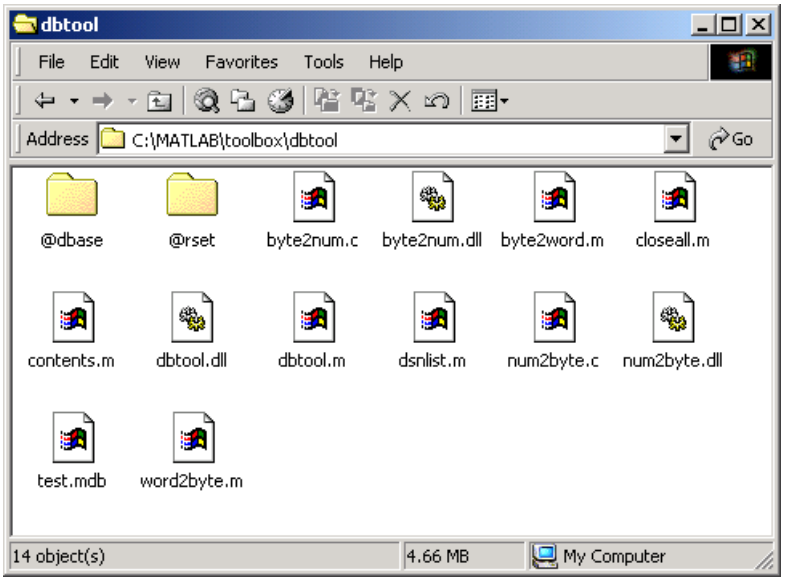

Figure 2.1: Install demo version of DBTool

2.1.2 Update MATLAB path

Now we add the DBTool path to the MATLAB search path. First click menu `File->Set Path...`, the `Set Path` dialog opens as seen in figure 2.2.

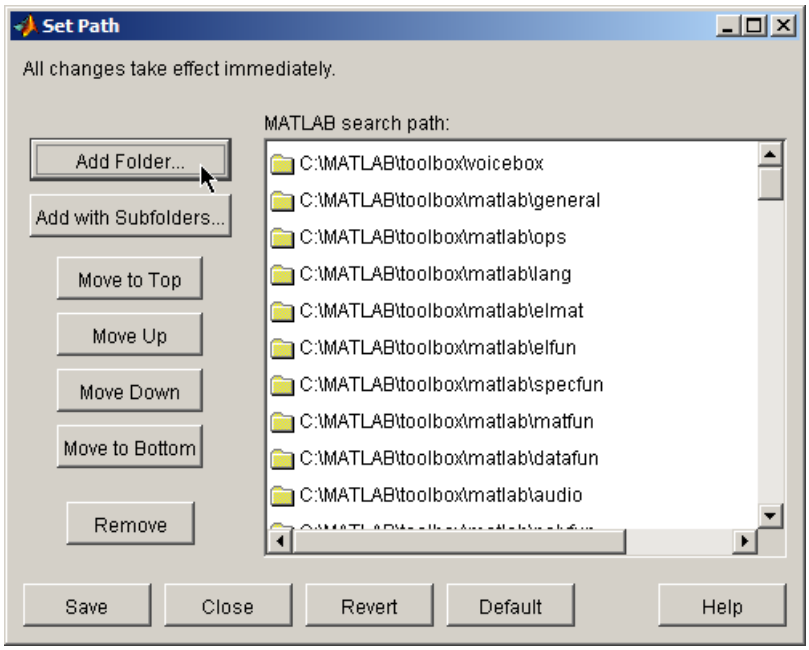

Figure 2.2: The Set Path dialog

Click `Add Folder...` button, and a folder selection dialog shows up, as seen in figure 2.3.

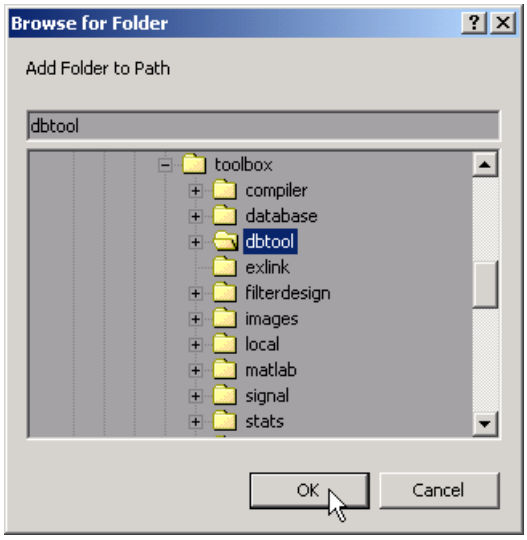

Figure 2.3: Select the DBTool folder

Select the folder `dbtool`, and click `OK`, then the `Add Folder...` dialog has the `dbtool` folder on the top of the list, as seen in figure 2.4.

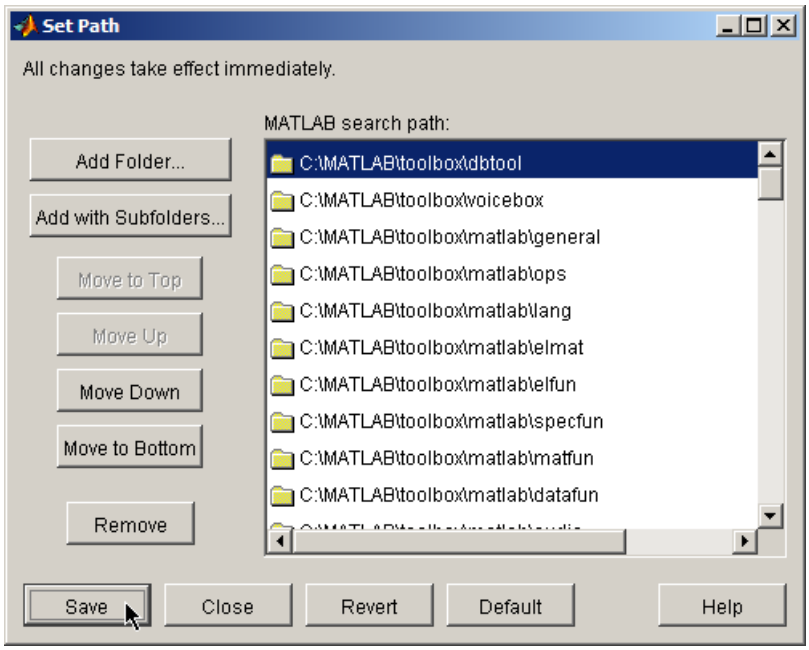

Figure 2.4: DBTool folder added to MATLAB search path

Click `Save` and then `Close`, and the setup of MATLAB search path is finished.

2.2 Licensed version installation

The standard, professional and redistributable versions of DBTool are provided in another `dbtool.zip`, which has the Plain API functions included. You need a license file named `license.dat` to activate the licensed version.

2.2.1 Unpack dbtool.zip

Unpack all the files in `dbtool.zip` to a folder using WinZIP, just as the same as installing the demo version.

2.2.2 Copy the license file

The license file `license.dat` will be emailed to you as an attachment. Just copy the license file into the folder of DBTool, as seen in figure 2.5.

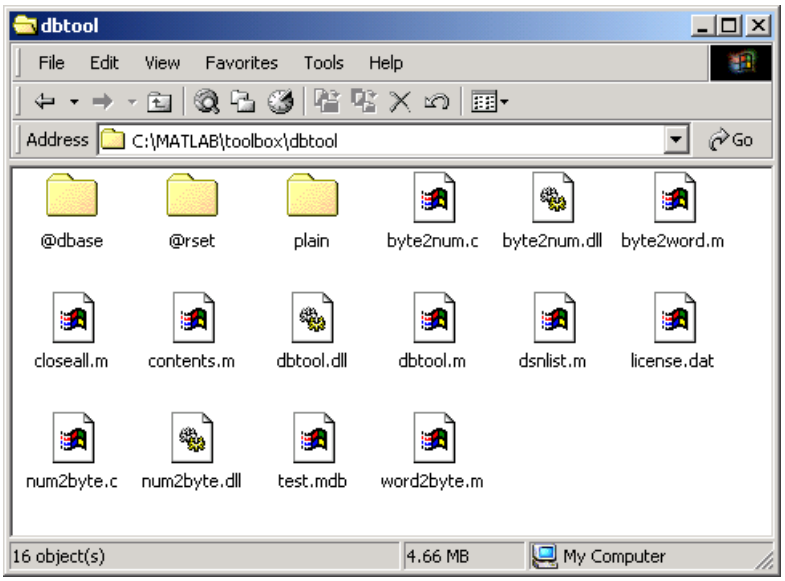

Figure 2.5: Install non-demo version of DBTool

2.2.3 Update MATLAB path

Add the DBTool path to the MATLAB search path, just likes installing the demo version. Be aware that licensed version has a folder `'plain'` which contains the Plain API functions, you also need to add the folder `'plain'` to the MATLAB path, if your want use Plain API, especially to compile your database program into a standalone executable with MATLAB Compiler.

2.3 Setup ODBC data source

2.3.1 Prepare to setup ODBC

Click the icon of `My Computer` and open it, find the `Control Panel` icon and open it. If you are running Windows 9x, you can find the `ODBC` icon there. If you are running Windows 2000/XP, you should first click the icon `Administrative Tools` in it, as seen in figure 2.6.

Figure 2.6: The Control Panel window

Open the `Administrative Tools` window, and find the `Data Sources (ODBC)` icon, as seen in figure 2.7.

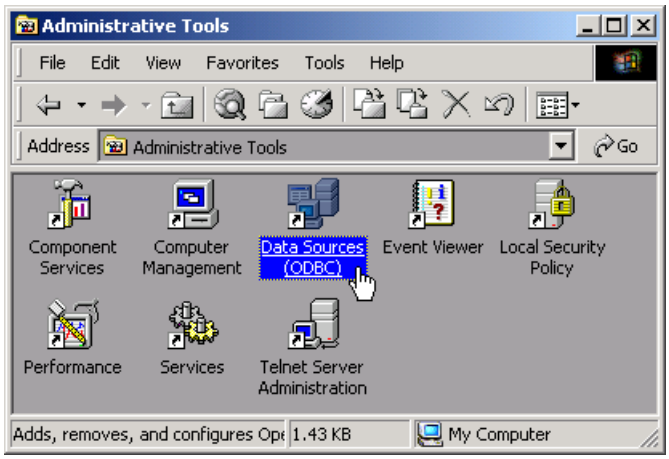

Figure 2.7: The Administrative Tools window

Click the ODBC icon and begin to setup the data sources, the `ODBC Data Source Administrator` window shows up, as seen in figure 2.8

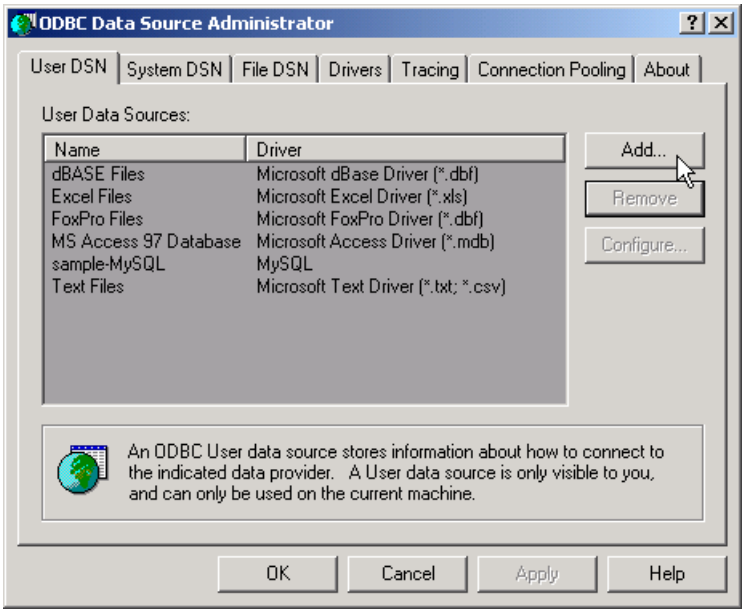

Figure 2.8: The ODBC Data Source Administrator window

2.3.2 Setup an Access ODBC data source

You can setup any kind of data sources. In this example, we demonstrate how to setup an Access data source.

First click the `ADD` button, in the `Create New Data Source` window, select `Driver do Microsoft Access(*.mdb)` from the ODBC driver list, and click `Finish`, as seen in figure 2.9.

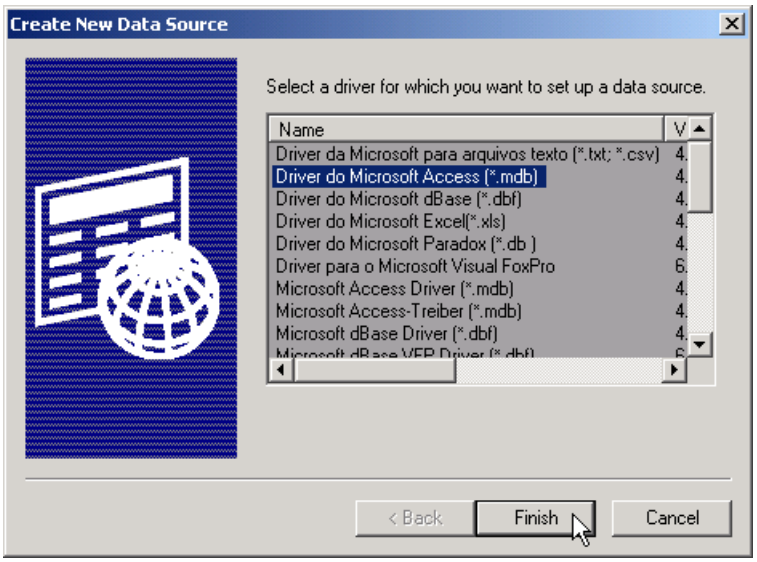

Figure 2.9: The Create New Data Source window

In the `ODBC Microsoft Access Setup` dialog, type in the Data Source Name, in this example, we use `testaccess`, as seen in figure 2.10.

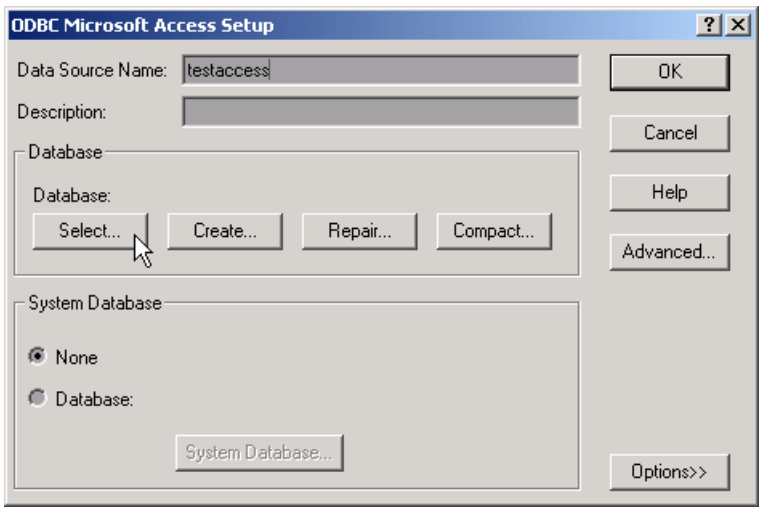

Figure 2.10: The ODBC Microsoft Access Setup dialog

Then click the `Select...` button, to select an access file (*.mdb). In this example, we select the file `c:\matlab\toolbox\dbtool\test.mdb`, as seen in figure 2.11.

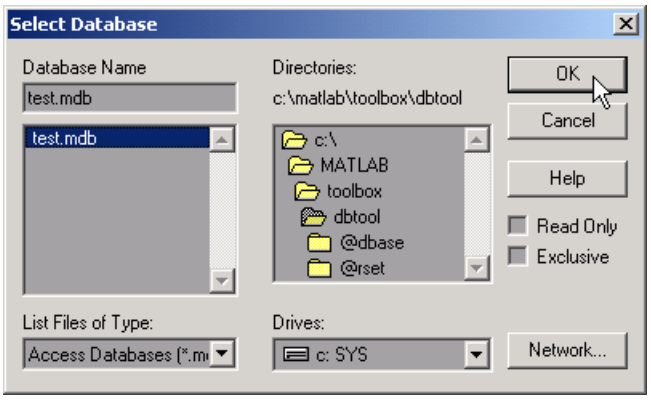

Figure 2.11: Select an Access file

After selected an Access file, the `ODBC Microsoft Access Setup` dialog looks like the figure 2.12.

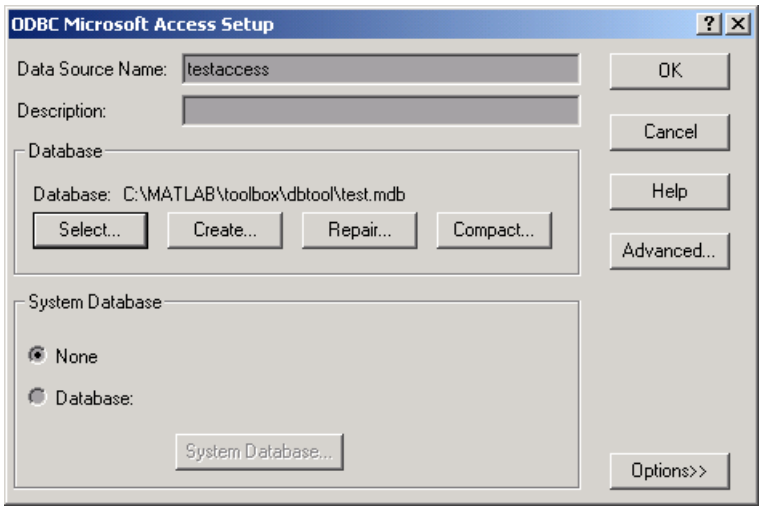

Figure 2.12: ODBC Microsoft Access Setup dialog is finished

After that, click `OK`, and the ODBC data source `testaccess` is finished,

2.3.3 Setup an MySQL ODBC data source

Now let's setup a MySQL ODBC data source. Before that, you need to install MySQL and MyODBC, which can be downloaded from http://www.MySQL.com/downloads/index.html.

After MySQL and MyODBC are installed and setup correctly, we need to create a database for test purpose. This can be done in MySQL command window, or use the GUI MySQL Administration tool: WinMySQLadmin.

Assume you have created a database named `test`, then you can setup a data source with it. First click `Add` button in the `Create New Data Source` window (figure 2.8).

In the `Create New Data Source` window, select `MySQL` in the driver list, and click `Finish`, as seen in figure 2.13.

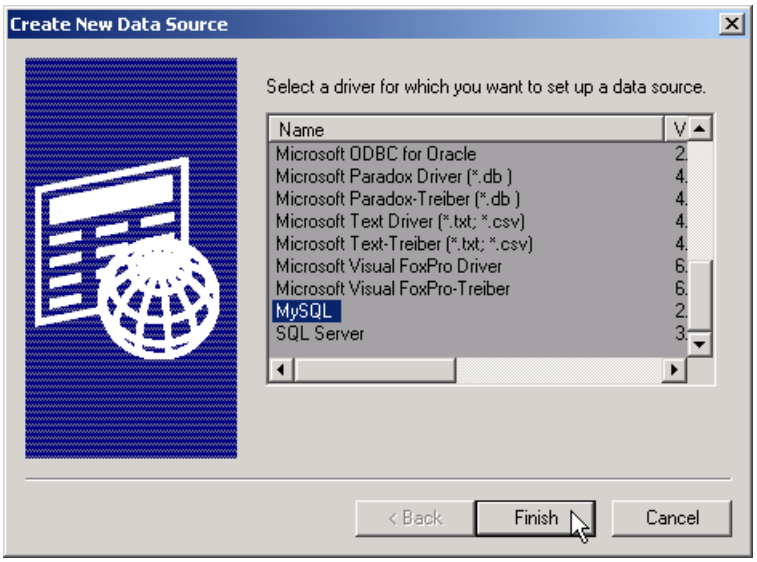

Figure 2.13: Add a MySQL ODBC data source

Then the MySQL ODBC data source configuration window shows up, as in figure 2.14. Type in the DSN name, for example: `testmysql`, and the MySQL server host (name/IP): `localhost`, or your IP, or domain name, and the MySQL database name: `test`, which is mentioned above.

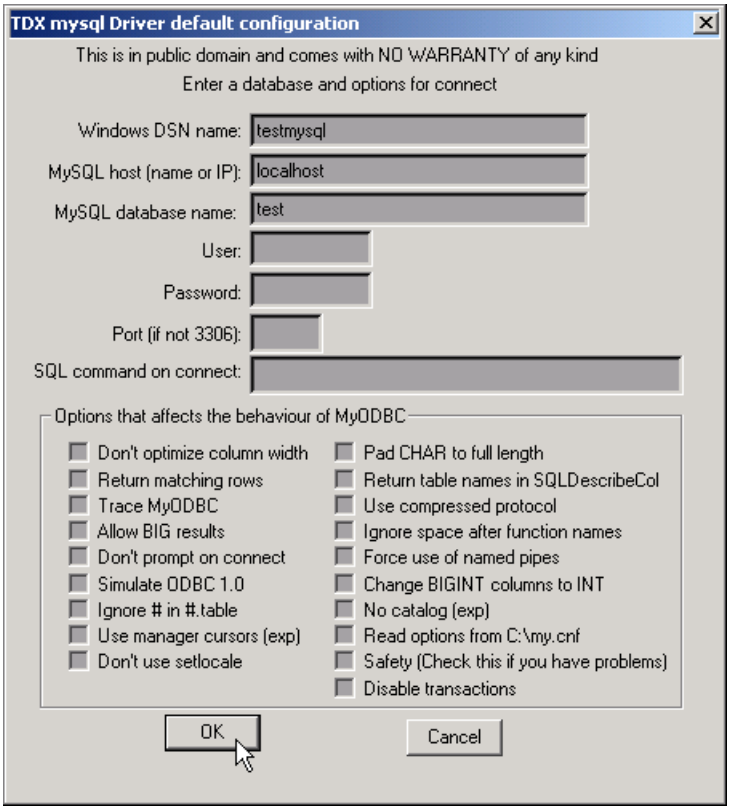

Figure 2.14: MySQL ODBC data source configuration window

After that, click `OK` to return to the `ODBC Data Source Administrator` window, as shown in figure 2.15. We can see that two data sources have been added: `testaccess` and `testmysql`.

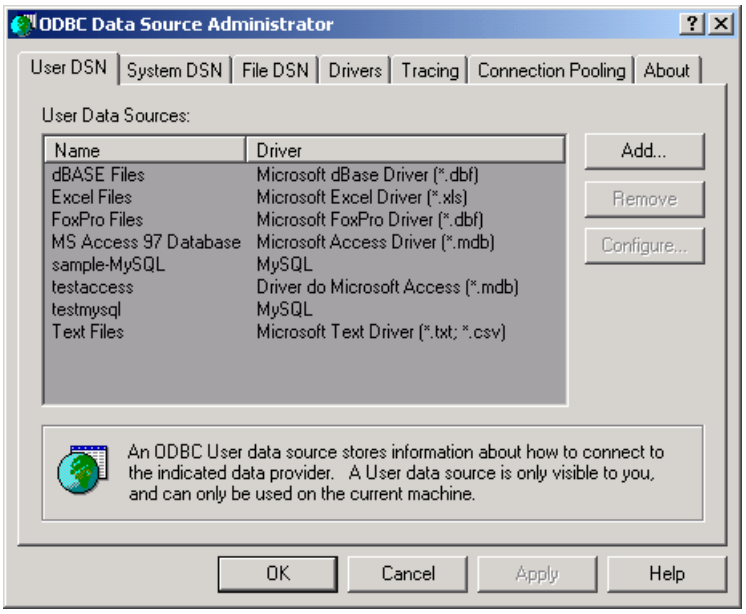

Figure 2.15: ODBC Setup with 2 new data sources

2.3.4 Setup ODBC data source for MATLAB Web Server

If you are running MATLAB Web Server to provide HTTP service using MATLAB and DBTool, don't create your ODBC Data Source as User DSN, use System DSN instead, which is the second tab in figure 2.8.

Chapter 3

Getting Started With DBTool

3.1 Test the installation

We'll test the installation by opening a database. Start MATLAB and type the command:

```
db=dbase('testaccess','','')
```

Where `testaccess` is the DSN name of Access ODBC data source we just setup, the following two empty strings are the user name and password separately. In this case, we don't need login authentication, so they can be empty.

If the installation is OK, we'll get the following display in the MATLAB command window:

```
>>db=dbase('testaccess','','')
Database object members:

Data source name: testaccess
       User name:
        Password:
          handle: 1

>>
```

Notice the last line "`handle: 1`" means the database is opened and is assigned a handle (=1) to the `dbase` object `db`. Of course, if the statement is followed by a ";", there will be on output.

If the installation is not correct, or the user has assigned an error DSN, user name or password, or any other errors happens, there should be some error message printed, and the returned db is empty. You can use `isempty` to test whether the operation is successful. For example, we open a DSN named `foo` which is not exist:

```
>> db=dbase('foo','','')
Warning: Data source name not found and no default driver specified

Failed to open database.

db =
```

```
    []

>>
```

3.2 Open a recordset

We have successfully opened a database, and the database information is saved in the `dbase` object `db`. Now type the following command:

```
rs=rset(db, 'select * from mytable')
```

This will open a recordset, where `db` is the `dbase` object, and the following string is a SQL command. In this case, the SQL command `'select * from mytable'` selects all fields in the table `mytable`. Of course, you can use any other valid SQL commands to select different fields with some special conditions.

This command will produce the following output:

```
>> rs=rset(db,'select * from mytable')

Recordset object members:
            handle: 1
       field count: 7
handle of database: 1
connect sql string: select * from mytable

field names:
    'ID'    'Name'    'Sex'    'Age'    'City'    'Date'    'Photo'

field types:
    'long'    'string'    'string'    'short'    'string'    'date'    'blob'

>>
```

The object `rs` has some properties, `handle` is the handle of the recordset, `field count` indicates there are 7 fields selected in the table, `handle of database` indicates the handle of the corresponding `dbase` object. `connect sql string` shows the SQL command string of this query. The following is `field names` and `field types` for all the fields.

If the statement is followed by a ";", there will be on output. The output is produced by the object `rs`.

3.3 Navigating in the recordset

Although a SQL query can return several rows of data, `rset` in the old version of DBTool was designed to fetch only one row of data each time. This is because when the table is very large, reading in the whole table is a waste of memory and time, especially when the data source is on another machine of the network, it may take a long time to load the whole table via the network. On the other hand, through the ODBC API, there is no way to find the total number

of rows in a recordset directly. But since DBTool version 2.0, multi-rows fetching is available, this is implemented by calling movenext internally in `dbtool.dll`, until the last row is reached or enough rows has been collected, and it's pretty faster than calling movenext in .m program.

A method is needed to access different rows for the object `rset`, this is called navigating. We can use the following 4 commands to navigate in the recordset.

- `movefirst` — Moves to the first row.
- `movelast` — Moves to the last row.
- `movenext` — Moves to the next row.
- `moveprev` — Moves to the previous row.

For example, we use the `movenext` to navigate to the second row, because when the recordset is opened, it's indicated to the first row.

```
>> movenext(rs)

ans =

     1

>>
```

By using `movenext` (or `moveprev`) continuously, you can navigate through all the rows of the recordset. When `movenext` (or `moveprev`) returns a 0, this means the last (or the first) row has already been reached, navigating should then stop.

3.4 Reading data from recordset

Once a `rset` object is successfully opened, we can fetch data from it. There are two methods to read data from the `rset` object:

- `fields` — Read and arrange the data into a structure.
- `fieldc` — Read and arrange the data into a cell.

The `fields` method returns a structure, its field names are the field names of the recordset, and its values are the values of the recordset. For example,

```
>> xs=fields(rs)

xs =

       ID: 1
     Name: 'Mike'
      Sex: 'male'
      Age: 25
     City: 'New York'
     Date: 7.3123e+005
    Photo: [1x5099 double]

>>
```

Now you can use `xs.Name`, `xs.Age`, etc. directly.

The `fieldc` method returns a cell, no field name information is included, only the field values are saved in sequence. For example,

```
>> xc=fieldc(rs)

xc =

    [1]    'Mike'    'male'    [25]    'New York'    [7.3123e+005]

    [1x5099 double]

>>
```

Now you can use `xc{1}` for field `ID`, `xc{3}` for field `Age`, and so on.

Adding a parameter '0' to read all rows into an array of structure or cell array:

```
xc=fieldc(rs,0);
xs=fields(rs,0);
```

Or specify the number of rows expected to read:

```
xc=fieldc(rs,5);
xs=fields(rs,5);
```

3.5 Close database and recordset

To close `dbase` and `rset` object, just use the `close` method. For example:

```
>> close(rs);
>> close(db);
```

Be sure not to close `dbase` objects which have `rset` objects opened, else there will be a warning message.

If you forgot which objects were opened, just use the command `closeall`:

```
>> closeall
```

This will close all opened `rset` and `dbase` objects.

In addition, when MATLAB is terminated, all `rset` and `dbase` objects are closed automatically (this feature is depended on the ODBC driver).

Chapter 4

Using DBTool

4.1 Editing row

To edit an existing row, we should use the `update` method with the following format:

```
update(rs, xc);
```

This will update the data of the current row using cell array `xc`.

Notice that only cell array is supported in updating, structure is not supported. The field values are stored in the cell `xc`, the number and sequence of the fields in `xc` must be the same as in the recordset.

For example,

```
>> db=dbase('testaccess','','');
>> rs=rset(db,'select Name,Age from mytable');
>> xc=fieldc(rs)

xc =

    'Mike'    [25]

>> xc{2}=26;
>> update(rs,xc);
>> xc=fieldc(rs)

xc =

    'Mike'    [26]

>> close(rs);
>> close(db);
>>
```

Multi-rows updating is not supported.

4.2 Inserting row

Use `insert` method to insert rows into the recordset. `insert` method is used in the following format:

```
insert(rs,xc);
```

For example, we copy the data of the first row, change the name and age, and insert it back into the recordset.

```
>> db=dbase('testaccess','','');
>> rs=rset(db,'select Name,Sex,Age,City,Date,Photo from mytable');
>> xc=fieldc(rs);
>> xc{1}='Bill';
>> xc{3}=22;
>> insert(rs,xc);
>> close(rs);
>> rs=rset(db,'select * from mytable where Name=''Bill''');
>> xc=fieldc(rs)

xc =

  Columns 1 through 6

    [49]    'Bill'    'male'    [22]    'New York'    [7.3123e+005]

    [1x5099 double]

>> close(rs);
>> close(db);
>>
```

In this example, the `ID` of the new inserted row is 49, but not 7, this is because `ID` is the key and index field, and the ODBC driver maintaines its value automatically.

Multi-rows inserting is also supported, prepare your data as a 2-dimentional cell array, for example, a 2 rows cell array with 3 fields can be constructed as:

```
>> xc = {'Nike' , 'male' , 24; 'Windy' , 'female' , 22}

xc =

    'Nike'     'male'      [24]
    'Windy'    'female'    [22]

>>
```

and then use the same command to insert rows.

```
insert(rs,xc);
```

Note: in demo version, multi-rows insertion is only functional for 30 minutes each time.

4.3 Deleting row

Use `delete` method to delete the current row from the recordset. It's used in the following format:

```
delete(rs);
```

For example, we delete the row inserted in the previous section.

```
>> db=dbase('testaccess','','');
>> rs=rset(db,'select * from mytable where Name=''Bill''');
>> delete(rs);
>> xc=fieldc(rs)

xc =

     []     ''     ''     []     ''     []     []

>> close(rs);
>> close(db);
>>
```

4.4 Execute SQL directly

When a database is opened, you can use `execsql` method to execute SQL command or stored procedure directly. The calling convention is:

```
execsql(db,'sql command');
```

For example, the following command create a new table named `pet` in the database, and then drop it.

```
>> db=dbase('testaccess','','');
>> execsql(db,'create table pet (name CHAR(20), birth DATE)');
>> execsql(db,'drop table pet');
>> close(db);
>>
```

To verify the creating and dropping of the new table pet, open the file `test.mdb` in Access to check it.

4.5 Date/Time field

We have seen the `Date` field of the table is a type of date/time, but when we display the data in `xs` or `xc`, there is only a number:

```
>> xs=fields(rs)

xs =
```

```
          ID: 1
        Name: 'Mike'
         Sex: 'male'
         Age: 25
        City: 'New York'
        Date: 7.3123e+005
       Photo: [1x5099 double]

>>
```

How to read the time stored in the field `xs.Date`? We can use the MATLAB command `datastr` to convert it into a string:

```
>> datestr(xs.Date)

ans =

10-Jan-2002 14:44:00

>>
```

Similarly, use MATLAB command `datenum` to convert time into number. And the current time can be obtained by MATLAB command `now`.

4.6 BLOB field

4.6.1 What is BLOB

DBTool supports BLOB fields. BLOB is binary large object. Using BLOB, user can store large binary or text data with variable length. The data stored as BLOB can be managed by user, or by the database manager, such as Access. In Access, BLOB fields are called Packages, which can store OLE objects or embed files directly into the table.

4.6.2 Reading file from BLOB field of Access database

The table `mytable` in database `test.mdb` has a field named `Photo`, which is a Package. Every `Photo` field is embedded with a .gif file. There is no difference to read the data from the BLOB fields than other fields. For example:

```
>> db=dbase('testaccess','','');
>> rs=rset(db,'select * from mytable');
>> xs=fields(rs)

xs =

          ID: 1
        Name: 'Mike'
         Sex: 'male'
```

```
      Age: 26
     City: 'New York'
     Date: 7.3123e+005
    Photo: [1x5099 double]

>>
```

We can see `xs.Photo` is a 1x5099 double array, each double word only contains a byte. Actually, it has a picture file named `Abra.gif` embedded. The file `Abra.gif` is 1165 bytes, other bytes are OLE information inserted by Access.

We can find the file content of `Abra.gif` by searching the string "GIF8", this is the beginning string of an ordinary GIF file.

```
>> pack = xs.Photo;
>> offset = findstr(pack, 'GIF8')

offset =

   184

>>
```

And the 4 bytes before the string "GIF8" is the length of the file:

```
>> pack(184-4:184-1)

ans =

   141     4     0     0

>>
```

So the length of the file can be calculated:

```
>> len=141+256*4

len =

        1165

>>
```

Now we can extract the content of the file, and save it to disk:

```
>> dat = pack(offset:offset+len);
>> fout = fopen('temp.gif','wb');
>> fwrite(fout, dat);
>> fclose(fout);
>>
```

Now we can open the file `temp.gif` by the Windows Explorer, or read it into MATLAB and plot it.

```
>> [xxx map] = imread('temp.gif');
>> imshow(xxx,map);
>>
```

The GIF file extracted from the BLOB field is shown in figure 4.1.

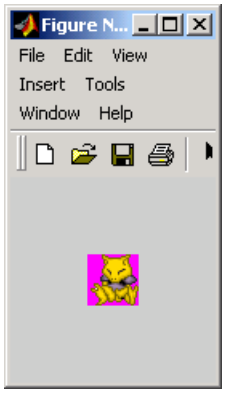

Figure 4.1: Show GIF extracted from BLOB

4.6.3 Storing double array in Access

In the Access test file `test.mdb`, another table named `arrays` is provided. It has the following fields:

- `'name'` - The name of the matrix to be saved, text.
- `'rows'` - Number of rows of the matrix, numeric.
- `'cols'` - Number of columns of the matrix, numeric.
- `'data'` - Stores binary data of the matrix, set to OLE Object at design, and changes to long binary automatically after data is written in.

In the following example, we first generate a random matrix `x`, convert it into byte series using the mex program `num2byte`, and construct a cell array `xc` contains the name and dimension information and the data in byte series. Then insert a new row into the table `arrays` . Finally read it back into a structure `xs`, and restore the data of the matrix from `xs.data` using mex program `byte2num`, and `reshape` it back into the original dimension.

```
>> db=dbase('testaccess','','');
>> rs=rset(db,'select * from arrays');
>> xc=fieldc(rs)

xc =

    ''    []    []    []

>> x=rand(3,4)
```

```
x =

    0.4447    0.9218    0.4057    0.4103
    0.6154    0.7382    0.9355    0.8936
    0.7919    0.1763    0.9169    0.0579

>> xc{1}='x';
>> xc{2}=3;
>> xc{3}=4;
>> xc{4}=num2byte(x);
>> insert(rs,xc);
>> xs=fields(rs)

xs =

    name: 'x                    '
    rows: 3
    cols: 4
    data: [1x96 double]

>> y=byte2num(xs.data);
>> y=reshape(y,xs.rows,xs.cols)

y =

    0.4447    0.9218    0.4057    0.4103
    0.6154    0.7382    0.9355    0.8936
    0.7919    0.1763    0.9169    0.0579

>> close(rs);
>> close(db);
>>
```

4.6.4 Writing and reading BLOB data into MySQL table

In this section, we give an example of accessing BLOB data with MySQL.

First create a table named `arrays`, with a field `name` to save the array's name and a field `data` to save the array's binary data, and the field `id` is used as the key.

```
>> db=dbase('testmysql','','');
>> execsql(db,'create table arrays (id int(10) default 0 not null,
               name char(20), data blob, primary key (id))');
```

Now insert a new row into the table manually.

```
>> execsql(db,'insert into arrays (id,name,data) values (0,''abc'',0)');
```

Then open a recordset and write data into the table.

```
>> rs=rset(db,'select * from arrays', 1);
>> xc=fieldc(rs)

xc =

    [0]    'abc'    [48]

>>
```

The `data` field is 48, this is because '0' is regarded as a character, not a number, and the ASCII code of '0' is 48 in decimal. Now we change the content of `xc` and update the data in the recordset, and then close it:

```
>> data=fix(127+127*sin((1:512)/512*2*pi));
>> xc{3}=data;
>> update(rs,xc);
>> close(rs);
```

To verify our modification is correct, try the following code:

```
>> rs=rset(db,'select * from arrays');
>> xc=fieldc(rs)
>> xc

xc =

    [0]    'abc'    [1x512 double]

>> plot(xc{3})
>> close(rs);
>> close(db);
```

Make sure to add a parameter '1' to open MySQL recordset in snapshot mode. See figure 4.2 for the plot of `xc{3}`.

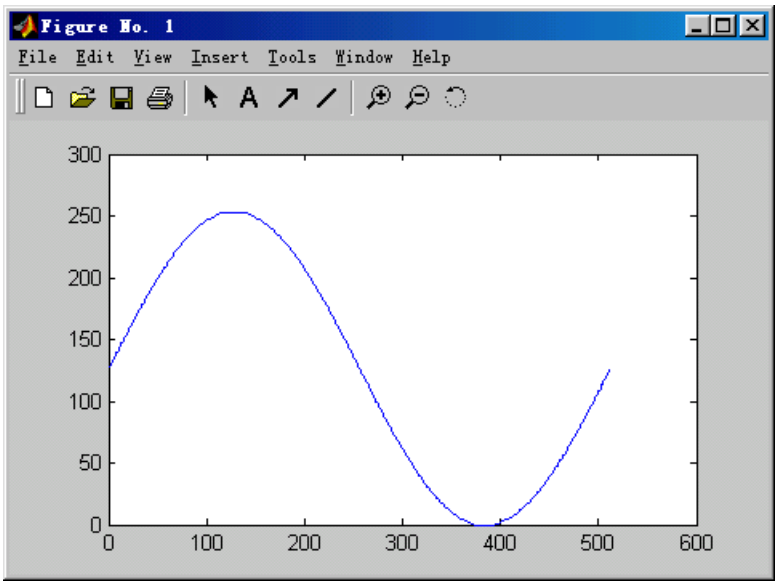

Figure 4.2: Data extracted from the BLOB field

Chapter 5

Using Plain API

5.1 What's Plain API

In the previous chapters, class `dbase` and `rset` for database operations are introduced. Using these two classes, one can write object oriented database program. But MATLAB Compiler does not support classes, so the program can't be compiled into standalone executable.

A set of Plain API functions is provided, which has no class objects and compatible with MATLAB Compiler. At the same time, the Plain API has very similar grammar to the class version. These include database operations:

- `db_open` - open dadabase
- `db_close` - close the database
- `db_execsql` - execute a sql string directly

and recordset operations:

- `rs_open` - open recordset
- `rs_close` - close recordset
- `rs_fields` - fetch data into structure array
- `rs_fieldc` - fetch data into cell array
- `rs_insert` - insert a new row
- `rs_delete` - delete current row
- `rs_update` - update current row
- `rs_movefirst` - move to the first row
- `rs_movenext` - move to the next row
- `rs_moveprev` - move to the previous row
- `rs_movelast` - move to the last row

For detailed description, see reference.

5.2 Work with MATLAB Compiler

5.2.1 Prepare your program

To make your program compatible with MATLAB Compiler, write your MATLAB program as a function, not a script file. This is simply by adding 'function' to the first line. In the following example `dbmcc.m`, a database structure `db` is first opened with `db_open`, and a recordset `rs` is opened with `rs_open`. Then use `rs_movefirst` and `rs_movelast` to navigate in the table, and use `rs_fields` and `rs_fieldc` to read data into a structure or a cell. Finally use `rs_close` and `db_close` to close database connections.

```
function dbmcc

fprintf('open db\n')
db=db_open('testaccess','','');

fprintf('open rs\n')
rs=rs_open(db,'select * from mytable');

fprintf('move first\n')
rs_movefirst(rs);

fprintf('read struct\n')
xs=rs_fields(rs)

fprintf('move last\n')
rs_movelast(rs);

fprintf('read cell\n')
xc=rs_fieldc(rs)

fprintf('close rs\n')
rs=rs_close(rs);

fprintf('close db\n')
db=db_close(db);
```

5.2.2 Compile into Standalone Executable

In MATLAB window, type `mcc -m foo.m` or `mcc -p foo.m` to compile your MATLAB program into a standalone executable. With the `'-m'` or `'-x'` directive, mcc will search all .m files called by the main program, and compile them into C or C++ files, and finally link them into a .exe file. Then you can run the file in a DOS prompt. If your program used GUI, use `mcc -B sgl foo.m` instead. For more information of `mcc`, type `help mcc` in MATLAB window.

Be sure to place a copy of `dbtool.dll` to the same folder of your main program. Otherwise mcc will not work properly. The following example is to compile `dbmcc.m` into `dbmcc.exe`.

```
>> mcc -m dbmcc.m                    %compile it into standalone
>> ls                                %dbmcc.exe appears
```

```
.                 dbmcc.h                 rs_fetch.c        rs_movelast.c
..                dbmcc.m                 rs_fetch.h        rs_movelast.h
db_close.c        dbmcc_main.c            rs_fieldc.c       rs_open.c
db_close.h        dbtool.dll              rs_fieldc.h       rs_open.h
db_open.c         dbtool_mex_interface.c  rs_fields.c
db_open.h         dbtool_mex_interface.h  rs_fields.h
dbmcc.c           rs_close.c              rs_movefirst.c
dbmcc.exe         rs_close.h              rs_movefirst.h
```

Now you can run dbmcc.exe in a DOS prompt, or just run it inside MATLAB, with a '!' ahead of the command.

```
>> !dbmcc                        %run dbmcc.exe, '!' means a DOS shell execution
open db
open rs
move first
read struct

xs =

       ID: 1
     Name: 'Mike'
      Sex: 'male'
      Age: 26
     City: 'New York'
     Date: 7.3123e+005
    Photo: [1x5080 double]

move last
read cell

xc =

  Columns 1 through 6

    [6]    'Susan'    'female'    [27]    'Pittsburgh'    [7.3122e+005]

  Column 7

    [1x4375 double]

close rs
close db
>>
```

5.3 Work with MATLAB COM Builder

5.3.1 Prepare your program

The following example `dbaccess.m` is a MATLAB GUI program which direct you to open a database, select a table and then show the contents of the first row.

```
function dbaccess

closeall;

dsns = dsnlist;
v = 0;
while v~=1
    [s,v]=listdlg('PromptString','Select a database:',...
                  'SelectionMode','single','ListString',dsns);
end
dsn = dsns{s};

db  = db_open(dsn,'','');
tbs = db_tablelist(db);
v = 0;
while v~=1
    [s,v]=listdlg('PromptString','Select a table:',...
                  'SelectionMode','single','ListString',tbs);
end
tb = tbs{s};

s = sprintf('Now let''s read the first row of\ntable "%s"
             from database "%s"', tb, dsn);
uiwait(msgbox(s,'DBAccess','modal'));

rs = rs_open(db,['select * from ' tb]);
xc = rs_fieldc(rs);
names = rs_fieldname(rs);
types = rs_fieldtype(rs);

str = '';
for i=1:length(xc)
  switch types{i}
    case {'bool','short','long','single','double'}
                        s = sprintf('%s : %d\n', names{i}, xc{i});
    case 'date';
                        s = sprintf('%s : %s\n', names{i}, datestr(xc{i}));
    case {'char','string'}
                        s = sprintf('%s : %s\n', names{i}, xc{i});
    case 'blob'
                        s = sprintf('%s : BLOB filed\n', names{i});
    otherwise,
```

```
                    s = '';
  end
  str = [str s];
end

rs_close(rs);
db_close(db);

s = sprintf('The first row of\ntable "%s" from database "%s"
            is:\n%s', tb, dsn, str);
uiwait(msgbox(s,'DBAccess','modal'));
```

By typing `dbaccess`, a listbox with all system DSN lists pops up.

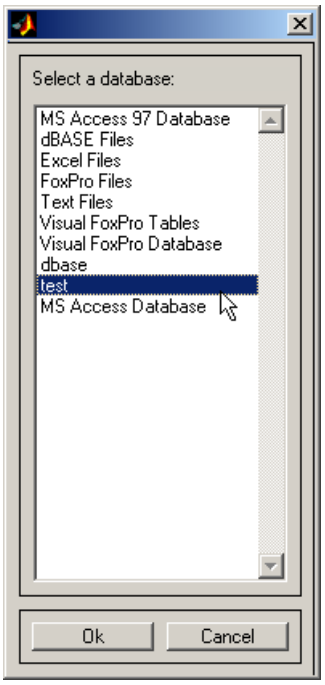

Figure 5.1: DBAccess: DSN list dialog

By selecting a DSN name, a `dbase` object is opened and another listbox pops up with the table lists in this database.

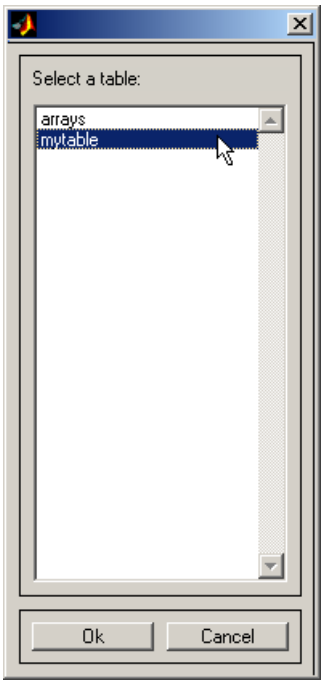

Figure 5.2: DBAccess: Table list dialog

Select a table in the list, and the contents of the first row is displayed in a dialog box.

Figure 5.3: DBAccess: message box

Figure 5.4: DBAccess: contents of first row

The function name is `'dbaccess'`, and we'll build a COM component `'DBTool_Demo'`, which has a method named `'dbaccess'`.

5.3.2 Compile into COM component

Before building your COM component with `comtool`, type this command to register `mwcomutil.dll` in a DOS prompt:

```
mwregsvr mwcomutil.dll
```

In MATLAB window, type `comtool` to envoke MATLAB COM Builder. Create a new project by clicking menu `File | New Project`, fill items like figure 5.5.

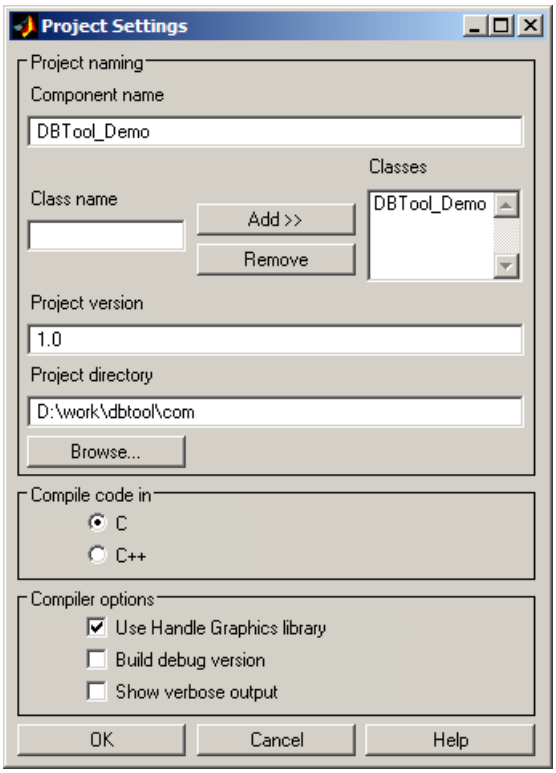

Figure 5.5: Create a new COM project

In the `Project Files` tree, add the file `dbaccess.m` into `M-files`. Every M-File you added becomes a method of the COM component. Then click the button `'Build'`, after a while, the COM object is compiled, the filename is `DBTool_Demo_1_0.dll`, in the folder `'distrib'`. To distribute the object, click menu `Component | Package Component` and you get a `'DBTool_Demo.exe'` in the `distrib` folder. This is a installer program with necessary MATLAB COM components and the `DBTool_Demo` component.

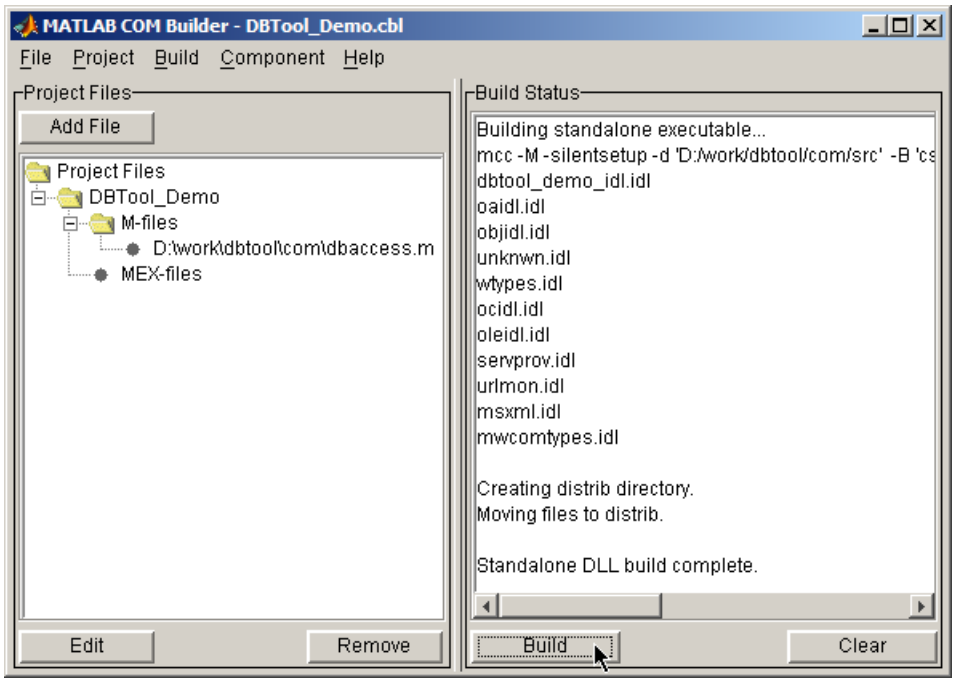

Figure 5.6: Add M-files and compile

Before testting the COM component, make sure the mex file `'dbtool.dll'` is in the system PATH. For more information on COM programing, refer to the manual of MATLAB COM Builder.

Chapter 6

References

6.1 dbase

The class object `dbase` is a database object. It can be opened by assigning an ODBC data source. Then several methods are used to operate on the tables in the database.

6.1.1 Properties

The properties of class object `dbase` can't be accessed directly. The only way to read it is by using the method `display`.

6.1.1.1 handle

Use the `handle` property to distinguish between different `dbase` objects.

6.1.1.2 dsn

The `dsn` property is the data source name in a string.

6.1.1.3 uid

The `uid` property is the user login name of the database. In many cases `uid` is not needed and an empty string `''` is used.

6.1.1.4 pwd

The `pwd` property is the user login password of the database. In many cases `pwd` is not needed and an empty string `''` is used.

6.1.2 Methods

6.1.2.1 dbase

Create an ODBC database object and open it.
Calling convention:

```
db=dbase(dsn,uid,pwd,options);
```

Inputs:

```
dsn     - string of data source name
uid     - string of username
pwd     - string of login password
options - optional, default to 8. Other values:
            2 : Open database read only
            4 : Use ODBC cursor lib
            8 : Don't display ODBC Connect dialog, default
           16 : Always display ODBC connect dialog
          Sum up the options needed, or leave blank to use default 8
          For example, use ODBC cursor lib and don't display ODBC Connect
          dialog, then options should be 4+8=12
```

Return:

```
db  - dbase object
```

Example:

```
db = dbase('testaccess','','');
db = dbase('testaccess','','',12);
```

If the database open operation not successful, the function returns `[]` (empty), you can use `isempty` to verify that.

6.1.2.2 close

Close an ODBC database.
Calling convention:

```
ret=close(db)
```

Input:

```
db - a database object
```

Return:

```
 1 - Success
 0 - Failure, maybe already closed.
-1 - Failure, not all recordsets closed.
```

6.1.2.3 tablelist

Show the table list in the current database.
Calling convention:

```
tb = tablelist(db, type, fmt);
```

Input:

```
db   - dbase object
type - Optional, default to 0, only list 'Table'
       0: Table only
       1: Table and View
       2: Table and System Table
       3: Table, View and System Table
fmt  - Optional string, output format, default to 'name'
       'name': return a cell array of table names
       'full': return a struct array of table name, type and owner
```

Return:

```
tb - cell array of table names
  or
tb - structure array of tables, with 3 fields:
     Name : table name
     Type : table type, 'Table', 'View' or 'System Table'
     Owner: table owner
```

6.1.2.4 settimeout

Set the timeout parameter in seconds. Default timeout is 15 seconds.
Calling convention:

```
settimeout(db, timeout);
```

Input:

```
db      - dbase object
timeout - timeout in seconds, default to 15, 0 = no timeout
```

6.1.2.5 execsql

Execute a SQL string.
Calling convention:

```
ret=execsql(db,sql);
```

Input:

```
db  - dbase object
sql - string of SQL command
```

Return:

```
1 - Success
0 - Fail
```

6.1.2.6 display

Display ODBC database members.
Calling convention:

```
display(db)
```

or

```
db
```

Input:

```
db - dbase object
```

Return:

```
Print the properties of the database object.
```

6.2 rset

6.2.1 Properties

The properties of class object `rset` can't be accessed directly. The only way to read it is by using the method `display`.

6.2.1.1 handle

The property `handle` is the handle of the recordset and used to distinguish between different `rset` objects.

6.2.1.2 hdb

The property `hdb` is the handle of the associated `dbase` object.

6.2.1.3 sql

The property `sql` is the SQL command issued to create the recordset.

6.2.1.4 field

The property `field` is a cell array to store the data of a row of the recordset.

6.2.1.5 fieldname

The property `fieldname` is a cell array of string to store the field name of the row.

6.2.1.6 fieldtype

The property `fieldtype` is a cell array of string to store the field type name of the row. Available field types are:

- bool
- char
- short
- long
- single
- double
- date
- string
- blob

6.2.1.7 fieldcount

The property `fieldcount` is the number of the fields.

6.2.2 Methods

6.2.2.1 rset

Creates a `rset` object from the `dbase` object `db` and SQL string `sql`.
Calling convention:

```
rset = rset(db,sql,type,options,blobsize);
```

Input:

```
db       - dbase object
sql      - database connect string
type     - open type, see below (optional)
options  - open options, see below (optional)
blobsize - set the maximum size of BLOB/MEMO fields in KB(optional)
```

Return:

```
rset object
```

Example:

```
rset = rset(db,'select * from mytab', type, options);
rset = rset(db,'select * from mytab');
rset = rset(db,'select * from mytab', 1);
rset = rset(db,'select * from mytab', [], [], 2048);
```

Open Type:

```
0 - dynaset     , uses SQLExtendedFetch, keyset driven cursor, default
1 - snapshot    , uses SQLExtendedFetch, static cursor
2 - forwardOnly , uses SQLFetch
3 - dynamic     , uses SQLExtendedFetch, dynamic cursor
```

Open Options:

```
0x0000 - none
0x0004 - readOnly
0x0008 - appendOnly
0x0010 - skipDeletedRecords, default
         Turn on skipping of deleted records, Will slow Move(n).
0x0020 - noDirtyFieldCheck
         Disable automatic dirty field checking
0x0100 - useBookmarks
         Turn on bookmark support
0x0800 - useExtendedFetch
         Use SQLExtendedFetch with forwardOnly type recordsets
0x2000 - executeDirect
         Directly execute SQL rather than prepared execute
```

Choose all options needed and add them up, convert it to decimal. For example, the default `skipDeletedRecords` is 0x10, and in decimal is 16. For detailed information, see the description of CRecordset::Open in Visual C++ Documentation of MSDN.

If the recordset open operation not successful, the function returns `[]` (empty), you can use `isempty` to verify that.

6.2.2.2 close

Close the recordset.
Calling convention:

```
ret=close(rs);
```

Input:

```
rs - rset object
```

Return:

```
1 - Success
0 - Fail
```

6.2.2.3 fields

Return field data in a structure array.
Calling convention:

```
data=fields(rs, rows);
```

Input:

```
rs   - rset object
rows - optional, max rows to read, default to 1, use 0 to read all following rows
```

Return:

```
data - rset object fields in structure
```

6.2.2.4 fieldc

Return field data in a cell array.
Calling convention:

```
data=fieldc(rs, rows);
```

Input:

```
rs   - rset object
rows - optional, max rows to read, default to 1, use 0 to read all following rows
```

Return:

```
data - rset object fields in cell
```

6.2.2.5 movefirst

Move to the first row.
Calling convention:

```
ret = movefirst(rs);
```

Input:

```
rs   - rset object
```

Return:

```
1 - Success
0 - Fail
```

6.2.2.6 movelast

Move to the last row.
Calling convention:

```
ret = movelast(rs);
```

Input:

```
rs   - rset object
```

Return:

```
1 - Success
0 - Fail
```

6.2.2.7 movenext

Move to the next row.
Calling convention:

```
ret = movenext(rs);
```

Input:

```
rs   - rset object
```

Return:

```
1 - Success
0 - Fail
```

6.2.2.8 moveprev

Move to the previous row.
Calling convention:

```
ret = moveprev(rs);
```

Input:

```
rs   - rset object
```

Return:

```
1 - Success
0 - Fail
```

6.2.2.9 movenext

Move to the next row.
Calling convention:

```
ret = movenext(rs);
```

Input:

```
rs   - rset object
```

Return:

```
1 - Success
0 - Fail
```

6.2.2.10 insert

Insert new row(s) into the recordset.
Calling convention:

```
ret=insert(rs,data);
```

Input:

```
rs   - rset object
data - rset data fields in cell array. Structure not supported
```

Return:

```
1 - Success
0 - Fail
```

6.2.2.11 update

Edit and update the current row in the recordset.
Calling convention:

```
ret=update(rs,data);
```

Input:

```
rs   - rset class
data - rset data fields in cell. Structure not supported
```

Return:

```
1 - Success
0 - Fail
```

6.2.2.12 delete

Delete current row in the recordset.
Calling convention:

```
ret=delete(rs);
```

Input:

```
rs   - rset object
```

Return:

```
1 - Success
0 - Fail
```

6.2.2.13 display

Display rset class members.
Calling convention:

```
display(rs)
```

or

```
rs
```

Input:

```
rs - rset object
```

Return:

```
Print the properties of the rset object.
```

6.2.2.14 isempty

Test recordset for empty.
Calling convention:

```
yn = isempty(rs);
```

Input:

```
rs - rset object
```

Return:

```
yn - 1: rs is empty, 0: rs is not empty.
```

6.3 Plain API

6.3.1 Database API

6.3.1.1 db_open

Create an ODBC database structure and open the database.
Calling convention:

```
db = db_open(dsn, uid, pwd, options);
```

Input:

```
dsn - string of data source name
uid - string of username
pwd - string of login password
options - optional, see below
```

Return:

```
db  - dbase structure
```

Example:

```
db = db_dbase('testaccess','','');
db = db_dbase('testaccess','','',12);
```

6.3.1.2 db_close

Close an ODBC database.
Calling convention:

```
db = db_close(db);
```

Input:

```
db - a dbase structure
```

Return:

```
Check db.handle,
0 - Success
1 - Failure, maybe already closed.
```

6.3.1.3 db_execsql

Execute a SQL string.
Calling convention:

```
ret = db_execsql(db,sql);
```

Input:

```
db  - dbase structure
sql - string of SQL command
```

Return:

```
1 - Success
0 - Fail
```

6.3.1.4 db_settimeout

Set query timeout in seconds.
Calling convention:

```
db_settimeout(db, timeout);
```

Input:

```
db  - dbase structure
timeout - timeout in seconds, default to 15, 0 = no timeout
```

6.3.1.5 db_tablelist

Get a structure array of table list in the database.
Calling convention:

```
tb = db_tablelist(db, type, fmt);
```

Input:

```
db   - dbase structure
type - Optional, default to 0, only list 'Table'
       0:Table only, 1: Table and View,
       2: Table and System Table, 3: Table, View and System Table
fmt  - Optional string, output format, default to 'name'
       'name': return a cell array of table names
       'full': return a struct array of table name, type and owner
```

Return:

```
tb - cell array of table names
  or
tb - structure array of tables, with 3 fields:
      Name : table name
      Type : table type, 'Table', 'View' or 'System Table'
      Owner: table owner
```

6.3.2 Recordset API

6.3.2.1 rs_open

Creates a rset structure from the dbase structure db and SQL string sql.
Calling convention:

```
rs = rs_open(db,sql,type,options,blobsize);
```

Input:

```
db       - dbase structure
sql      - database connect string
type     - open type (optional)
options  - open options (optional)
blobsize - max size of BLOB and MEMO fields (optional)
```

Return:

```
rs - rset structure
```

6.3.2.2 rs_close

Close the recordset.
Calling convention:

```
rs = rs_close(rs);
```

Input:

```
rs - rset structure
```

Return:

```
Check rs.handle,
0 - Success
1 - Failure, maybe already closed.
```

6.3.2.3 rs_fields

Return field data in a structure array.
Calling convention:

```
data = rs_fields(rs, rows);
```

Input:

```
rs   - rset structure
rows - optional, max rows to read, default to 1, use 0 to read all following rows
```

Return:

```
data - rset structure fields in structure
```

6.3.2.4 rs_fieldc

Return field data in a cell array.
Calling convention:

```
data = rs_fieldc(rs, rows);
```

Input:

```
rs   - rset structure
rows - optional, max rows to read, default to 1, use 0 to read all following rows
```

Return:

```
data - rset structure fields in cell
```

6.3.2.5 rs_insert

Insert new row(s) into the recordset.
Calling convention:

```
ret = rs_insert(rs,data);
```

Input:

```
rs   - rset structure
data - rset data fields in cell array. Structure not supported
```

Return:

```
1 - Success
0 - Fail
```

6.3.2.6 rs_delete

Delete current row in the recordset.
Calling convention:

```
ret = rs_delete(rs);
```

Input:

```
rs - rset structure
```

Return:

```
1 - Success
0 - Fail
```

6.3.2.7 rs_update

Edit and update the current row in the recordset.
Calling convention:

```
ret = update(rs,data);
```

Input:

```
rs   - rset class
data - rset data fields in cell. Structure not supported
```

Return:

```
1 - Success
0 - Fail
```

6.3.2.8 rs_movefirst

Move to the first row.
Calling convention:

```
ret = rs_movefirst(rs);
```

Input:

```
rs - rset structure
```

Return:

```
1 - Success
0 - Fail
```

6.3.2.9 rs_movenext

Move to the next row.
Calling convention:

```
ret = rs_movenext(rs);
```

Input:

```
rs - rset structure
```

Return:

```
1 - Success
0 - Fail
```

6.3.2.10 rs_moveprev

Move to the previous row.
Calling convention:

```
ret = rs_moveprev(rs);
```

Input:

```
rs - rset structure
```

Return:

```
1 - Success
0 - Fail
```

6.3.2.11 rs_movelast

Move to the last row.
Calling convention:

```
ret = rs_movelast(rs);
```

Input:

```
rs - rset structure
```

Return:

```
1 - Success
0 - Fail
```

6.3.2.12 rs_isempty

Test whether a recordset is empty.
Calling convention:

```
ret = isempty(rs);;
```

Input:

```
rs - rset structure
```

Return:

```
1 - rs is empty
0 - rs is not empty
```

6.4 Utilities

6.4.1 dbwarn

Enable/disable DBTool warning messages.
Calling convention:

```
dbwarn(yn);
```

Input:

```
yn - 1: enable warning messages(default), 0: disable waring messages.
```

6.4.2 dblasterr

Get last error/warning message from the dbtool mex file. Enable/disable DBTool warning messages.
Calling convention:

```
s = dblasterr();
```

Return:

```
s - last error/warning message string
```

6.4.3 dsnlist

Show DSN list. The user and/or system DSN names can be retrieved simply by typing this command.
Calling convention:

```
dsns = dsnlist(type);
```

Input:

```
type - Optional, default to 'usr', only list user DSNs
    'usr': only user DSNs
    'sys': only system DSNs
    'all': both user and system DSNs
```

Return:

```
dsns - cell array of DSN names
```

6.4.4 word2byte

Convert 16-bit signed word series into unsigned byte series. The lower byte is first, and the higher byte is the second. For example, (0x1234 0x2345 0x3456 0x4567) is converted into (0x34 0x12 0x45 0x23 0x45 0x34 0x67 0x45). This function is used when storing 16-bits signed word array into BLOB fields of the table.
Calling convention:

```
y=word2byte(x);
```

Input:

```
x - 1xN signed word array, range in [-32768, +32767].
```

Return:

```
y - 1x2N unsigned byte series.
```

6.4.5 byte2word

Convert unsigned byte series into 16-bit signed word series. The lower byte is first, and the higher byte is the second. For example, (0x34 0x12 0x45 0x23 0x45 0x34 0x67 0x45) is converted into (0x1234 0x2345 0x3456 0x4567). This function is used when reading unsigned byte series from BLOB fields of the table and restoring 16-bits signed word array from it.
Calling convention:

```
y=byte2word(x);
```

Input:

```
x - 1x2N unsigned byte series.
```

Return:

```
y - 1xN signed word array, range in [-32768, +32767].
```

6.4.6 num2byte

Convert double array into unsigned byte series. Because a double number is represented by 8 bytes, this function unpack the 8 bytes of the elements in the double array. This function is used before storing double arrays into BLOB fields of the table.
Calling convention:

```
y=num2byte(x);
```

Input:

```
x - double array, in any dimention, could be 1xN, Nx1, MxN, MxNxP, etc. .
```

Return:

```
y - 1xN unsigned byte series. The length of y is 8 times of N,
    which is the total number of elements in x.
```

6.4.7 byte2num

Convert unsigned byte series back into 1xN double array, where N is 1 of 8 of the size of input. This function is used to restore double arrays from BLOB fields of the table. Since the dimension information is not saved int the BLOB fields, a reshape function must be used to restore dimension.
Calling convention:

```
y=byte2num(x);
```

Input:

```
x - 1xN double array contains unsigned char values 0~255.
```

Return:

```
y - 1xN/8 double array restored.
```