DirectODBC v1.60
Delphi ODBC Components

by

Guernsey Software Company
Dan Daley
Mariah Hinkley

RecordNo

Events
- Connection events
  - BeforeCommit
  - BeforeConnect
  - BeforeDisconnect
  - BeforeRollback
- Query events
  - After/BeforeActivate
  - After/BeforeClose
  - After/BeforeDeleteRec
  - After/BeforeEditRec
  - After/BeforeFirst
  - After/BeforeInsertRec
  - After/BeforeLast
  - After/BeforeNext
  - After/BeforePrior
  - After/BeforeSave
  - After/BeforeScroll
  - After/BeforeUpdate
  - OnGet/OnSetFieldText
  - RowChanged
  - RowsAffected
  - UpdateError

ODBC Data Structures
- Date
- Time
- TimeStamp

Contacting Me
- Reporting bugs
- Requesting features
- Performance Issues

Notice

This is the non shareware version of DirectODBC. Each registered copy is to be used by a single developer at any one time. They are not to be redistributed in the form of Delphi components or as source code. They are for use in end-user applications and are to be distributed royalty free as part of a compiled, executable application.

In short, any person using this software in the form of Delphi components or possessing the source code should be a registered user of DirectODBC.

Introduction

Though I had several reasons to write these components, my main motivation was to provide direct access to ODBC for Delphi users who are using ODBC already. I wanted components that could connect through ODBC without adding the extra "layer" of IDAPI. In addition to communicating directly to ODBC I wanted to be able to edit result sets of robust queries. I wanted to be able to edit the result of any query my server (or at least my ODBC driver) could return. Finally, I wanted to be able to edit multiple records (or even multiple result sets) before any changes were sent to the server. That is, I wanted to work with a copy of a result set and send multiple update/delete/insert queries to reflect all of my edits as a batch.

I have accomplished these goals in this trial version. However, there is work still to be done. Sorting, searching, and

filtering of result sets (without having to issue more queries) and more data aware controls. Plus, any feedback from testers which I can incorporate.

In order to ensure that these components will work to your satisfaction, test them thoroughly and send as much feedback as possible. Enjoy.

Installation
1) Unzip DrctODBC.Zip into the desired directory
2) from Delphi choose Options|Install Components...
3) choose add from the install dialog, then choose browse
4) find the directory where you unzipped the files and choose RegODBC.PAS
5) choose OK
6) choose OK (at the install dialog)
7) they should be installed ok

Run the demo:
If you installed local interbase and can connect via the Interbase ODBC data source, then you should run the demo project. You will need to unzip demo.zip. Then open demo.dpr. First you should run the app and click the create table button. This will create a demo table and fill it with some data. Then, clicking any of the navigation buttons will open the query. You can edit the data in the edit controls. After you have edited a few rows, click the Update button (this will send your edits to the server).

Now close and re-run the app. Check that your edits worked.

Now, return to design mode and find the component called DemoQuery. Double click the SQL property. Notice, you were editing the result of a union...

Connecting to the database

Choosing a data source:
Connecting to a database requires that you have ODBC installed and configured. Assuming that you have ODBC setup, place an ODBCCon component on a form. The DataSource property should provide you with a list of available ODBC data sources. Choose the data source to which you wish to connect.

Connect Method:
If your data source only requires a user name and password for the connection, then you should leave ConnectionMethod set to cmConnect. Then you only need to provide a user name and password via the UserID and Password properties prior to connecting. However, if your data source requires more information to be passes via a connection string, then you should provide all of the information in the ConnectionStr property and set ConnectionMethod to cmDriverConnect.

NOTE: Currently the driver connect option uses Driver_Complete option during the connection call. This means that your users will be prompted by dialogs from the driver requesting missing connection information. The dialogs use the ConnectionStr string as defaults. Other driver connect options will be provided in the future.

Now you are ready to connect. You can now set active to true and the component will establish a connection to your data source.

Commit modes:
The default AutoCommit mode is True. This means each statement issued to the server will be commited for you. If you want to control when a transaction is committed, set AutoCommit to false. You will then be responsible for ending the transactions. A transaction will be started by the driver before you issue your first SQL statement.

NOTE: You are responsible for knowing the behaviour of your server regarding transactions and cursors. If you do not know how what the state of a cursor will be after ending a transaction, then you should only use the Query component with RetrieveAsNeed (see below for a description) set to false and should not prepare your queries.

Isolation levels:
The default isolation level is tiDefault. This means that the isolation level will be determined by your server. Currently, all possible values supported by ODBC are listed, regardless of whether or not they are supported by a particular data source. You are responsible for knowing which isolation levels are supported by your server. Future version will only list levels which are supported by the selected source.

Viewing tables:
Currently, the only way to view the available tables for a given data source is to use the Tables property on the ODBCCon component. Double clicking this property will bring up a dialog which will provide you with a list of tables (in a ComboBox) from the data source. Selecting a table in the ComboBox will provide you with information about the fields in the table.

Viewing procedures:
To view stored procedures on you server, double click the procedures property. This will present you with a dialog similar to the table information dialog. You can choose a stored procedure in the combobox and information about the procedures parameters will be displayed in the listbox. Not all ODBC drivers and/or databases support this option.

Issuing a Query

Choosing the data source:
You can issue a query against any data source for which there exists an ODBCCon component. You select the data source by setting the DataSource property for the ODBCQuery component. The drop down will provide you with a list of all the DataSource values of all of the ODBCCon components which are in existence. So to issue a query against a particular data source, you must have an ODBCCon component set to connect to that data source.

Setting the SQL Statement:
You can set the SQL property to any valid SQL statement that your ODBC driver will accept. For example, you can issue "select * from Customer" (without the quotes) against the local interbase employee database.

Setting parameter values and data types:
You can use parameters in your SQL statements by placing a question mark (?) in the query where you want a parameter (eg select * from Customer where Cust_No = ?). Before you can activate this statement, you must tell the component what data type this parameter is (not the data type you wish to set is as). For example, although you may want to set the above parameter from a string variable, the actual data type should be set to SQLInteger.

To set the parameter information at design time, double click the Params property. This will bring up a dialog to assist you in providing this information. You can set the following properties from the dialog: name, data type, parameter type, length, scale, value. Not all of the properties are valid for all data types. For example, length and scale are meaningless for an integer type.

Meaning of properties:
Name: the name to give to the parameter. This defaults to Paramx where x is the order of occurrence of the parameter in the statement.
Data Type: the data type the server will expect.
Parameter type: input or output
Length and Scale: depend on the data type
SQLChar,
SQLVarChar: length is the max # of char the field can hold. scale is meaningless
SQLNumeric,
SQLDecimal: length is the number of digits. scale is number of decimal places
SQLInteger, SQLSmallint, SQLFloat, SQLReal,
SQLDouble, SQLDate, SQLTime, SQLBigInt,
SQLTinyInt, SQLBit: length and scale are meaningless.
SQLBinary,
SQLVarBinary: length is the number of bytes of information. scale is meaningless
SQLTimeStamp: length is # of chars in the "yyyy-mm-dd hh:mm:ss[.fff[fff]]" format. scale is the # of digits in the fractional part of the timestamp.

SQLLongVarChar,
SQLLongVarBinary: length is max # of chars the field can hold. scale is N/A
Value: is a valid value for the parameter type. NOTE: currently the string representation of dates must be in the form of yyyy/mm/dd or yyyy-mm-dd and time must be in the form of hh:mm:ss. TimeStamps are Date:Time.Fraction.

Setting parameter info at run-time:
These properties may also be set at run-time. The corresponding property names are: ParamName, SQLType, ParamType, DataLen, and Scale. The value of a parameter may be set through the following properties: AsString, AsInteger, AsFloat, AsDate, AsTime, or AsTimeStamp.

To access a parameter at run-time, use the Params property of the Query component. This is an array property which is 1 based (ie Params[1] returns the first param). For example, to set the first parameter as a timestamp:
MyQuery.Params[1].AsTimeStamp := MyTimeStampStruct; (see data structures)

RetrieveAsNeeded option:
The retrieve as needed option tells the query component to only download the records from the server as they are request (via Next, Last). The default is false, which will download the entire result set when the query is activated. This allows the query to close the cursor (thus not leaving cursors open on the server).

Update information (for updatable queries):
The query component can update tables on the server for you, if you provide sufficient information. More than one table can be updated, however only one table per call to update will be updated. Therefore, to update more than one table, you must change the update information between calls to Update (to be discussed).

To be able to update a table, you must tell the query which table to update, which fields are updatable, and which fields are the key fields (fields to be used in the where clause to identify a record). If a field has been aliased in the select statement, you also need to set the Actual Field value on the update table information dialog (or alternatively the TableFieldName property on the field itself) to be the name of the field which is to be updated. NOTE: blob fields cannot be used as key fields.

At design time you can use the Update Table dialog to fill in this information. To activate the dialog, double click the UpdateTable property of your query component. This will provide you with a list of all of the fields returned by your query. First, type the name of the table to be udpated in the table name edit box. Then, select the fields to be used as key fields (to select a field, highlight it and click the > button). Select the fields that you want to be able to update in this table. Fill in display label and alias information for each field if desired and click OK. You are done.

At runtime (or design time) you can provide all of this information via the UpdateTable, EditFields, and KeyFields properties on the query and DisplayLabel and TableFieldName properties on the field objects. The EditFields and KeyFields are string lists and UpdateTable is a string.

NOTE: currently all of the fields returned by your statement are shown. So not all of these fields will necessarily be in your table. Be sure to only select fields from the table you want to update. At a later time, the field list will be filtered once you choose the table to update.

<u>Retrieving data from a query</u>

Reading data from field objects:
After activating a query, you can access data through field objects. Field objects provide data access through properties. Currently, most fields only allow data access through the AsString property. The only exceptions to this are the BinaryField and TextField (see below).

However, the following functions have been provided to assist you with data conversions:
function StringToDate(Value: String): Date_Struct;
function DateToString(Value: Date_Struct): String;
function StringToTime(Value: String): Time_Struct;

```
function TimeToString(Value: Time_Struct): String;
function StringToTimeStamp(Value: String): TimeStamp_Struct;
function TimeStampToString(Value: TimeStamp_Struct): String;
```

More data access properties will be provided in future versions.

The Field property of a Query is an 1 based array property. So to access the first field simply do the following

```
MyQuery.Field[1].AsString := DateToString(MyDate);
```

You can also access fields through the FieldByName function.

```
MyQuery.FieldByName('MyDateField').AsString := DateToString(MyDate);
```

Blob and Text Fields:
Blob/Binary fields allow data access through the AsString property as well as through ReadBlock and WriteBlock member functions. The AsString property will only return the first 255 bytes of data and is not the recommended method of access. The preferred methods of access are:

```
LoadFromStream(aStream: TStream);
SaveToStream(aStream: TStream);
LoadFromFile(FileName: String);
SaveToFile(FileName: String);
Truncate(NewSize: LongInt);
ReadBlock(Offset, Size: LongInt; Buffer: PChar; var BytesRead: LongInt);
WriteBlock(Offset, Size: LongInt; Buffer: PChar);
```

One method of reading blob data from a field is as follows:

```
Offset := 0;
ReadBlock(
    Offset,  { byte # to begin reading }
    BufferSize { size of your buffer },
    Buffer { your pchar buffer },
    BytesRead { # of bytes actually read into buffer }
);
while BytesRead > 0 do
begin
    { process data in buffer }
    Inc(Offset, BytesRead);
    ReadBlock(Offset, BufferSize, Buffer, BytesRead); { read the next block }
end;
```

Writing data to a blob is analogous to reading. To truncate the size of a blob, call Truncate with the size of the new blob. Use this if you write new blob data which is smaller than the existing data.

Text fields descend from Binary fields and also allow access through AsStrings, which returns a TStringList object.

Sending updates to the server/data source:
When you are ready to send your update to the server, simply call the Update method. This will issue all of the update, insert and delete statements to reflect your edits on the server data.

Handling problems during the update operation:
Two types of problems can occur during the update process: errors raised by the server (ODBC) or errors raised by the component. If an error is returned by ODBC then the UpdateError event is called (if defined) or handled in a default manner with an Abort, Retry, Ignore dialog (if not defined). After any update, delete or insert is called, the RowsAffected event is called. You can check in this event whether or not more than one row was effected by a statement (or if no rows were effected) and raise an exception accordingly.

NOTE: the problem record will be the active record when this event is called. However once the Update is finished, the

record which was active prior to the update will be made active again (even if you raise an exception in this event). So make note as to which record is the problem record if you need to.

You can try to re-issue your updates as many times as you like, prior to calling ResetRows. ResetRows will clear the record and field status flags for all records in the result set. This method should be called after calling Update successfully (if further edits are intended). You can reset the flags for a particular record by calling ResetRowFlags while that record is current.

To update more than one table from the current result set, you need to call update once for each table to be updated. Between calls to update you need to change UpdateTable, KeyFields, and EditFields to the appropriate values.

For Example:

```
try
    with ODBCQuery1 do
    begin
        Update;
        UpdateTable := 'SecondTable';
        with EditFields do
        begin
            Clear;
            Add('Field1');
            Add('Field2');
            Add('Field3');
        end;
        with KeyFields do
        begin
            Clear;
            Add('KeyField1');
            Add('KeyField2');
        end;
        Update; { updates a second table }
        MyConnection.Commit;
        ResetRows;
    end;
except
    MyConnection.Rollback; { if an error occurs, rollback }
    raise;
end;
```

Using data aware controls

To make accessing data easier you can use the data aware controls provided in this package. Currently, there are three controls for you to use: TODBCEdit, TODBCMemo, and TODBCGrid.

TODBCImage, TODBCEdit and TODBCMemo:

The image, edit and memo components can be linked to fields in your result set. To establish this link you need to set two properties: ODBCSource and ODBCField. ODBCSource can be either a TODBCQuery or TODBCSource component. Once you have set this property, you can drop down the ODBCField property editor and choose the field you want to edit. The list box will show a list of all fields available in the result set. The control is now linked to that field. Changes made in the control will be saved to the field when either the control loses focus or Save is called (Save is call for you when the record is scroll or Update is called).

NOTE: the image component is currently read only. To set the value assign to the underlying field object (see binary fields).

TODBCGrid:

The grid component allows you to edit the non-blob fields of a result set in a tabular format. To edit the data in a grid you

need to link the grid to an ODBCQuery component. This is accomplished by setting the ODBCSource property on the grid. As with the other data aware controls this property can be set as a TODBCSource or TODBCQuery.

By default the grid will display all the fields from the result set. To display a subset of the fields and/or to display the fields in a different order, you set the DisplayFields property. The DisplayFields property is a TStringList. To set this property simply put the name of each field that you want displayed on a separate line of the string list.

Formatting display data

To display data in the non-default manner, you can use the OnGetFieldText and OnSetFieldText of the TODBCQuery component. The OnGetFieldText event is called whenever a data aware control requests data (actually, whenever the AsText property of a TODBCField object is referenced). This event gives you the opportunity to manipulate the string that will be displayed. The default value is passed to the event as the Value parameter. If you want the data displayed differently, simply assign a string to Value.

If you are changing the default display format for a field, you will most likely need to convert the data before it can be stroed again. The OnSetFieldText event is called whenever a field is assigned to via the AsText property. The value to be assigned to the field is passed in the Value parameter. If you need to do any conversion on the data, simply re-assign the new value to this variable.

For example to display TRUE/FALSE for an integer field instead of 0 and 1 you could do the following (assuming the field name is MyBool):

```
procedure TForm1.MyQueryGetFieldText(AField: TODBCField; var Value: OpenString);
begin
    if AField.FieldName = 'MyBool' then
    begin
        if AField.AsInteger = 0 then Value := 'FALSE'
        else Value := 'TRUE';
    end;
end;

procedure TForm1.MyQuerySetFieldText(AField: TODBCField; var Value: OpenString);
begin
    if AField.FieldName = 'MyBool' then
    begin
        if Value = 'FALSE' then Value := '0'
        else Value := '1';
    end;
end;
```

Other useful query methods

Buffer:

The buffer property will let you change the active buffers. There are three possible buffers for you to traverse. They are the result set buffers (bData), deleted buffers (bDeleted), and the original data buffers (bOriginal). To change buffers assign the appropriate value to the buffer property (eg to view the deleted records, Buffer := bDeleted). Note: the deleted and original buffers are read only. To edit these you must call UnDelete or Undo (see below).

Undo:

The undo method can undo edits to the active record, even if the changes have been saved in the record buffer. That is, you have called Save or have scrolled off the record since the changes occured (This method will not undo changes to the database if Update has been called). To undo changes to a record, position to the appropriate record and call undo.

UnDelete:

UnDelete allows you to move a buffer from the deleted buffers to the result set buffers. To undelete a buffer you must have the deleted buffers active and must be positioned on the record to be undeleted. Then, call UnDelete.

RecordNo:
RecordNo is a read/write property. This is the index into the active buffers of the currently accessible record. You can assign to this property to move directly to a given record.

Events

Connection events:

BeforeCommit:
BeforeCommit is called when the Commit method is invoked. It is called prior to the commit being issued to ODBC. This event can be used to prevent the commit from being issued by raising an exception.

BeforeConnect:
BeforeConnect is called when a connection is attempted, but prior to calling the ODBC API connect routines. This provides the opportunity to prompt the user for connection information or other pre-connection initialization.

BeforeDisconnect:
This event is called when a disconnect is requested, but prior to the disconnect being issued.

BeforeRollback
BeforeRollback is called when the Rollback method is invoked. It is called prior to the rollback being issued to ODBC. This event can be used to prevent the rollback from being issued by raising an exception.

Query events:
Most of the ODBCQuery events occur in pairs. They are called in pairs when the appropriate action is requested. The before event occurs before any action transpires and the after event occurs after the activity is completed. Note that you can usually prevent any activity by raising an exception in the before event.

After/BeforeActivate:
Occur before and after the query is activated. That is when Active is set to true, these events are triggered.

After/BeforeClose:
Occur when Active is set to false.

After/BeforeDeleteRec:
Occur when a record is deleted.

After/BeforeEditRec:
Occur before a record is edited for the first time since Save was last called.

After/BeforeFirst:
Occur when First is called.

After/BeforeInsertRec:
Occur when a record is inserted. You can use the after insert event to initialize new records.

After/BeforeLast:
Occur when Last is called.

After/BeforeNext:
Occur when Next is called.

After/BeforePrior:
Occur when Prior is called.

After/BeforeSave:
Occur when the current record buffer is going to be saved to the list of record buffers.

After/BeforeScroll:
Occur when any scrolling takes place.

After/BeforeUpdate:
Occurs when Update is called.

OnGet/OnSetFieldText:
See Formatting display data.

RowChanged:
This event is called whenever a new record buffer is made active. This can happen when scrolling, saving, undo-ing, deleting, cancelling edits, inserting, etc. This event is called whenever all controls should refresh their data.

RowsAffected:
See Handling problems during the update operation.

UpdateError:
See Handling problems during the update operation.

Data and Class Structures

ODBC Structures:

```
 Date_Struct = record
   Year: integer;
   Month: word;
   Day: word;
 end;

 Time_Struct = record
   Hour: word;
   Minute: word;
   Second: word;
 end;

 TimeStamp_Struct = record
   Year: integer;
   Month: word;
   Day: word;
   Hour: word;
   Minute: word;
   Second: word;
   Fraction: longint;
 end;
```

Class Structures:
See classes.wri

Contacting Me:
I encourage any and all feedback on these components. I will read any correspondence regarding this work. I only ask that you follow a few simple guidelines when sending feedback. First, be verbose. Describe any problems or feature request in as much detail as possible. Include any information that you think may be valuable (date of your ODBC.DLL, manufacturer and version of your ODBC driver, the type and version of your server, the version of Delphi you are using, etc.). If I cannot reproduce your bug, or do not know what you are talking about, I cannot fix the problem. Second, include a method for me to contact you. If you failed in step one, I may need to contact you for more info. If you have an e-mail address, include that. Third, include in the subject/title reference to the type of feedback (if it is a bug report include

bug in the title; if it is a feature request, include request in the title; etc).

I can be contacted as follows:

Compuserve: 74211,3513
e-mail: daley@guernsey.com
url: http://www.guernsey.com
voice: (408) 335-3955

mail:
Dan Daley
2505 Lowell Ct.
Simi Valley, CA 93065