Contents

What is ODBC?

ODBC is an acronym for 'Open DataBase Connectivity'. It is a standard first developed by Microsoft, which is now widely used by all major database vendors. You get a lot of different databases today. Big databases driven by database engines include Oracle, Microsoft SQL Server, Sybase, Informix and DB2, while you also get a lot of so-called 'file-based' databases with no stand-alone database engines, such as Microsoft Access, dBase, Paradox and Btrieve.

Now you need a way to communicate with a database and transfer data to and from a database, as well as perform some queries on the data in the database. The problem is that each database vendor has his own set of methods (called an API) to perform all these functions. So, if you write an application which for example uses the Oracle API, that same application can't be used to access a Microsoft SQL Server database, since the API's of Oracle and Microsoft SQL Server are different. So you have to duplicate all your code if you want to access two different databases from the same application. Imagine if you want to access 5 or 10 different databases!

Now this is where ODBC comes into the picture and makes life a whole lot easier for you. ODBC is an API that allows you to talk to any database you want. Instead of writing a data access application using a specific database API, such as Oracle's, you can now write your application using the ODBC API, which can perform virtually any function that you can do with the Oracle API. The best part is that now you can use the same code to access any other database too! How is this possible? Well, ODBC is basically a very thin layer between your application and the database API. You get a large number of different ODBC drivers. Each ODBC driver maps each function in the ODBC API to similar functions in the API of the specific database the driver was written for. For example, the Oracle ODBC driver maps the ODBC API into the Oracle API.

How do you specify what ODBC driver must be used at any time by your application? This is what ODBC DataSources are for. An ODBC DataSource is used by your application to indicate which database it must use. But where do DataSources come from? You create a DataSource using your ODBC Administrator, usually located in your Windows Control Panel. If you are writing a 16bit application, you must make use of the 16bit ODBC Administrator. For 32bit applications you must make use of the 32bit ODBC Administrator (to distinguish it from the 16bit one, a yellow '32' is printed across the icon). When you load the ODBC Administrator, you will see a list of existing ODBC DataSources. To change an existing DataSource, click on the Setup button. To add a new DataSource, click on the Add button, select the ODBC driver for the specific database that you want to create a DataSource for, and click on the OK button. You now have to fill in the details required to exactly describe the database to be represented by the DataSource.

Now that we understand how ODBC work and what an ODBC DataSource is, lets see how much work it is to change the current database of your data access application which you have written using the ODBC API: To do it, change the current ODBC DataSource to the new ODBC DataSource. Is that all? Just one line of code? Now isn’t that a million times better than rewriting all your code just to make use of a number of different databases?

To give a brief description of the ODBC architecture, all ODBC interaction are done via three types of handles:

The Environment Handle is used to establish an ODBC environment for your application, by setting up a communication link from your application to the ODBC Driver Manager. You normally only need one of these handles in your application.

For each Environment Handle, you can have a number of Connection Handles in your application. Each Connection Handle gives you a different connection to a database. You can have multiple Connection Handles to the same database or multiple Connection Handles to multiple databases in your application. It is at this level that transactioning is done. Transactions done with each Connection Handle are isolated from transactions done with other Connection Handles.

For each Connection Handle, you can have a number of Statement Handles. These handles allow you to manipulate the data in your database. Because you can have multiple Statement Handles for each Connection Handle, each Statement Handle can perform a different function at the database and can be in a different state than the other Statement Handles.

After you have all the handles you need, you can use them together with the ODBC functions in the ODBC API to make calls to the ODBC Driver Manager. The ODBC Driver Manager routes all the ODBC calls you make to the correct ODBC Driver, to allow you to interact with the appropriate database. The ODBC Driver Manager also manages the loading and unloading of the ODBC drivers. People often confuse the function of the ODBC Driver Manager with that of the ODBC Administrator. While the ODBC Administrator allows you to manage your DataSources, it is not a required part of ODBC when you run your application. The ODBC Driver Manager on the other hand is vital to the execution of your ODBC-enabled application, and allows you to perform all the data-access operations you require, against any ODBC-enabled database.

What is ODBCExpress?

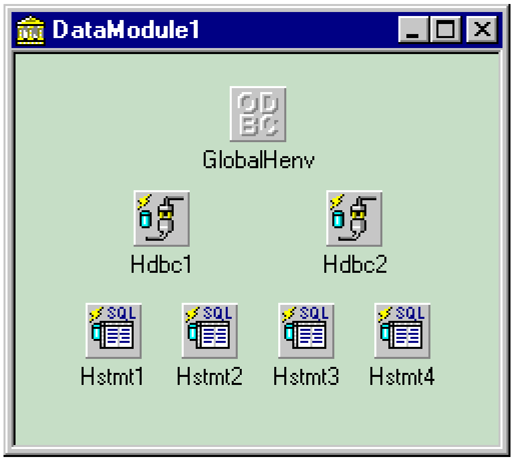
Even though you now only have to write to one API to access any database, the ODBC API isn’t a simple API by any means. It has to cater for almost any functionality that any database out there can support, so you can imagine that the ODBC API is quite complex. Now ODBCExpress wraps the ODBC API into a set of easy-to-use classes for Delphi and C++Builder, so that you don’t even have to know much (if anything) about the ODBC API to fully make use of ODBC in your applications!

So, what about a brief description of the ODBCExpress architecture? At the heart of ODBCExpress are three components, collectively called the ODBC Handle Components:

The Environment Handle Component, which is used to establish an ODBC environment for you applications.

The Connection Handle Component, which is used to establish a connection to a database, given a DataSource.

The Statement Handle Component, which is used to communicate with a database and query the data in the database.



As seen in the image above, the 3 handle components can be arranged in a tree structure for use in your application. You only need one Environment Component for your application, to which you can assign a number of Connection Components, each with its own connection to a database. So you can immediately see that you can connect to different databases from your application at the same time. For each Connection Component, you can also have a number of Statement Components, each performing its own functionality at the database. Because you only need one Environment Component, it's not a non-visual component as depicted above, but a global class variable called GlobalHenv, which is automatically created by ODBCExpress for your application.

Apart from these handle components, ODBCExpress also makes use of a number of Visual Data-Aware Controls and Non-Visual Components to easily manage, display and edit data from a database. The Visual Data-Aware Controls used by ODBCExpress are the normal Delphi, C++Builder and 3rd-party data-aware controls. The Non-Visual Components allow you to do things such as managing your DataSources and database schema, retrieving database catalog Information, and easily creating installation programs for your ODBC enabled applications.

ODBCExpress consists of the following two main levels of units:

At the bottom level is the ODBC Call-level Interface (OCI). The OCI consists of two units, OCIH.dcu and OCI.dcu. The OCI unit contains a Delphi mapping into the ODBC API and the OCIH unit contains all type and constant declarations used by the layers on top of the OCI.

The top and main level is the ODBC Class Library (OCL). The OCL consists of four units, OCL.dcu, OSI.dcu, OVCL.dcu and ODSI.dcu. The OCL unit contains all the ODBCExpress error classes and handle components, the OSI and OVCL units contains all the ODBCExpress visual controls and non-visual components, and the ODSI unit contains the ODBCExpress descendants of the virtual TDataSet in Delphi and C++Builder, which are used to tie in with the data-aware controls.

ODBCExpress supports ODBC 3.0 and above and its basic functionality will work with all ODBC drivers with at least a Level 1 ODBC API Conformance. However, to make use of the full ODBCExpress functionality, Level 2 or near-Level 2 ODBC drivers are recommended.

The ODBCExpress Web Site can be found at http://www.odbcexpress.com. At this web site you'll find the latest evaluation versions of ODBCExpress, documentation, example programs and tutorials on ODBCExpress, as well as other ODBC related materials such as ODBC documentation and ODBC drivers.

White Paper

The ODBCExpress White Paper is a very important addition to the topics discussed in this help file and is highly recommended reading material for ODBCExpress users.

The white paper provides Delphi and C++Builder users with a clear, high-level understanding of the Open Database Connectivity (ODBC) architecture as well as the context in which ODBCExpress fits into this architecture. The ultimate aim is to provide existing and potential users of ODBCExpress with a solid understanding of how ODBCExpress works.

Most Delphi or C++Builder users with little or no concept of the ODBC SQL-oriented programming model find it difficult to move from a non-SQL-oriented programming model such as the Borland Database Engine (BDE) to ODBCExpress. The white paper therefore also attempts to highlight the fundamental differences between ODBCExpress and the BDE.

The white paper, called "oewpaper.pdf", is shipped with ODBCExpress in PDF format and can be viewed and printed using the free Adobe Acrobat Reader.

SQL Operations

Data at a DataSource can be manipulated with ODBCExpress both through ODBC function calls and SQL operations. SQL operations forms a main part of ODBCExpress and it is therefore important that you have a good knowledge of how to perform SQL operations.

The ODBCExpress statement component is used to perform SQL operations at a DataSource. The following three diagrams demonstrate the main function sequences that can be performed with the statement component:

Diagram 1	Inserts, Updates and Deletes
Diagram 2	Simple Selects
Diagram 3	Complex Selects

SQL can also be used to perform any other function at the database, such as manipulation of databases, tables, views, indexes and stored procedures.

The SQL examples below (and other examples throughout the help file) all make use of the following table at a DataSource:

Students(SNo, SName, SGender, SBirth, SComment, SPic)

The columns in the table have the following data types:

SNo	String
SName	String
SGender	Boolean
SBirth	Date/Time
SComment	Text Blob
SPic	Binary Blob

Examples:

Example 1	Inserting rows into the table.
Example 2	Updating rows in the table.
Example 3	Deleting rows from the table.
Example 4	Selecting rows from the table without a condition.
Example 5	Selecting rows from the table with a condition.

Diagram 1

Inserts, updates and deletes can be done by setting and preparing an appropriate SQL statement once and binding all the parameters of the statement once. After that is done, values can be assigned to the parameters and the statement can be executed multiple times, as is illustrated with the following diagram:

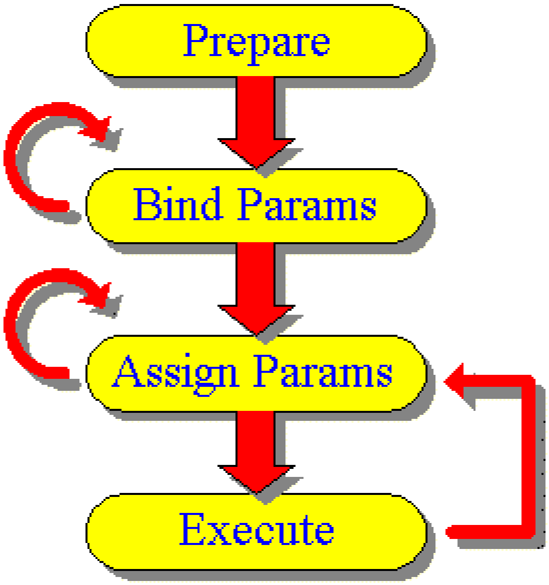


Diagram 2

A simple select can be done by setting, preparing and executing a SQL statement once to generate a result set. After that is done the rows in the result set can be fetched from the DataSource, as illustrated in the following diagram:

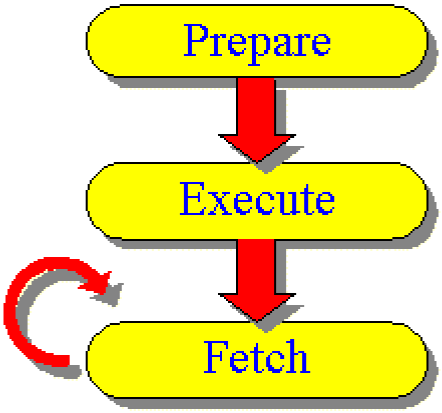
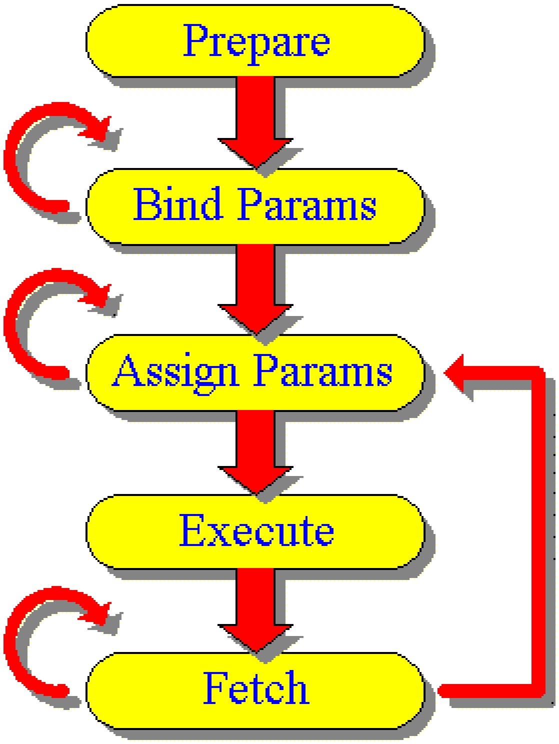


Diagram 3

A complex select can be done by setting, preparing and binding the parameters of a SQL statement once. After that is done, values can be assigned to the parameters and the statement executed to generate a result set at the DataSource. This last part can be done multiple times to generate different result sets, fetching the rows from the DataSource each time, as illustrated in the following diagram:



Insert Example

In this example, we'll insert a 100 rows at the DataSource:

Delphi Code

```
var
  i: Integer;
  //the NullString type is an array of characters
  SNo, SName: NullString;
  SGender: Byte;
begin
  with Hstmt1 do
  begin
    //set and prepare an insert statement, with three parameters, each indicated by '?'
    SQL:= 'INSERT INTO Students (SNo, SName, SGender) VALUES (?, ?, ?)';
    Prepare;

    //bind the variables declared to the three parameters
    BindNullString(1, SNo);
    BindNullString(2, SName);
    BindByte(3, SGender);

    for i:= 1 to 100 do
    begin
      //assign values to the variables
      StrPCopy(SNo, '97'+IntToStr(i));
      StrPCopy(SName, 'Student '+IntToStr(i));
      SGender:= i mod 2;

      //execute the statement to add a new row at the DataSource
      Execute;
    end;
  end;
end;
```

C++ Code

```
int i;
//the NullString type is an array of characters
NullString SNo, SName;
Byte SGender;
AnsiString s;

//set and prepare an insert statement, with three parameters, each indicated by '?'
Hstmt1->SQL = "INSERT INTO Students (SNo, SName, SGender) VALUES (?, ?, ?)";
Hstmt1->Prepare();

//bind the variables declared to the three parameters
Hstmt1->BindNullString(1, SNo);
Hstmt1->BindNullString(2, SName);
Hstmt1->BindByte(3, SGender);

for (i=1; i<=100; i++) {
  //assign values to the variables
  s = "97"+IntToStr(i);
  strcpy(SNo, s.c_str());
  s = "Student "+IntToStr(i);
  strcpy(SName, s.c_str());
  SGender = (Byte)(i % 2);

  //execute the statement to add a new row at the DataSource
  Hstmt1->Execute();
}
```

So why are we using the NullString datatype instead of the String or AnsiString data type for SNo and SName? The reason is that at the time a variable is bound to a SQL statement, the variable must have memory allocated for it. Because string variables are dynamic, memory is allocated for them only the first time that you assign a value to the string. Also, each time you assign a new value to the string, new memory may be allocated for the string, possibly at a different location in memory, making the bind you've previously made with the variable invalid. You can for example use string variables as follows:

Delphi Code

```
var
  i: Integer;
  SNo, SName: String;
  SGender: Byte;
begin
  with Hstmt1 do
  begin
    SQL:= 'INSERT INTO Students (SNo, SName, SGender) VALUES (?, ?, ?)';
    Prepare;

    //allocate memory for the string variables by setting their length before the binds
    SetLength(SNo, 255);
    BindString(1, SNo);
    SetLength(SName, 255);
    BindString(2, SName);
    BindByte(3, SGender);

    for i:= 1 to 100 do
    begin
      //copy the values into the strings to avoid memory re-allocation of the strings
      StrPCopy(PChar(SNo), '97'+IntToStr(i));
      StrPCopy(PChar(SName), 'Student '+IntToStr(i));

      //do not assign values directly to the strings as follows, because it might make the previous
binds invalid
      SNo:= '97'+IntToStr(i);
      SName:= 'Student '+IntToStr(i);

      SGender:= i mod 2;
      Execute;
    end;
  end;
end;
```

C++ Code

```
int i;
AnsiString SNo, SName;
Byte SGender;
AnsiString s;

Hstmt1->SQL = "INSERT INTO Students (SNo, SName, SGender) VALUES (?, ?, ?)";
```

```
Hstmt1->Prepare();

//allocate memory for the string variables by setting their length before the binds
SNo.SetLength(255);
Hstmt1->BindString(1, SNo);
SName.SetLength(255);
Hstmt1->BindString(2, SName);
Hstmt1->BindByte(3, SGender);

for (i=1; i<=100; i++) {
  //copy the values into the strings to avoid memory re-allocation of the strings
  s = "97"+IntToStr(i);
  strcpy(SNo.c_str(), s.c_str());
  s = "Student "+IntToStr(i);
  StrPCopy(SName.c_str(), s.c_str());

  //do not assign values directly to the strings as follows, because it might make the previous binds invalid
  SNo = "97"+IntToStr(i);
  SName = "Student "+IntToStr(i);

  SGender = (Byte)(i % 2);
  Hstmt1->Execute();
}
```

You can see that the first code example in which the NullString variables are used is somewhat simpler than the second code example in which the string variables are used. Of course if you are only going to insert one row, you can allocate memory for the string variables by assigning values to the strings before binding them:

Delphi Code

```
var
  SNo, SName: String;
  SGender: Byte;
begin
  with Hstmt1 do
  begin
    SQL:= 'INSERT INTO Students (SNo, SName, SGender) VALUES (?, ?, ?)';
    Prepare;

    //assign values to the string variables to force memory allocation for the strings
    SNo:= '9700001';
    SName:= 'Van der Merwe';
    SGender:= 1;

    //bind the variables
    BindString(1, SNo);
    BindString(2, SName);
    BindByte(3, SGender);

    //execute the statement to add the new row at the DataSource
    Execute;
  end;
end;
```

C++ Code

```
AnsiString SNo, SName;
Byte SGender;

Hstmt1->SQL = "INSERT INTO Students (SNo, SName, SGender) VALUES (?, ?, ?)";
Hstmt1->Prepare();

//assign values to the string variables to force memory allocation for the strings
SNo = "9700001";
SName = "Van der Merwe";
SGender = 1;

//bind the variables
Hstmt1->BindString(1, SNo);
Hstmt1->BindString(2, SName);
Hstmt1->BindByte(3, SGender);

//execute the statement to add the new row at the DataSource
Hstmt1->Execute();
```

Another way of specifying parameters in a SQL statement is by replacing each unnamed parameter indicated by a question mark "?" with a named parameter indicated by a colon ":". These named parameters can then be used during binding instead of specifying parameter positions:

Delphi Code

```
var
  SNo, SName: String;
  SGender: Byte;
begin
  with Hstmt1 do
  begin
    //set and prepare an insert statement with three named parameters ANo, AName and AGender
    SQL:= 'INSERT INTO Students (SNo, SName, SGender) VALUES (:ANo, :AName,
:AGender)';
    Prepare;

    SNo:= '9700001';
    SName:= 'Van der Merwe';
    SGender:= 1;

    //bind the named parameters using the BindByName methods - notice that order isn't important
    BindByteByName('AGender', SGender);
    BindStringByName('ANo', SNo);
    BindStringByName('AName', SName);

    Execute;
  end;
end;
```

C++ Code

```
AnsiString SNo, SName;
Byte SGender;

Hstmt1->SQL = "INSERT INTO Students (SNo, SName, SGender) VALUES (:ANo, :AName,
:AGender)";
Hstmt1->Prepare();

//assign values to the string variables to force memory allocation for the strings
SNo = "9700001";
SName = "Van der Merwe";
SGender = 1;

//bind the variables
Hstmt1->BindByteByName("AGender", SGender);
Hstmt1->BindStringByName("ANo", SNo);
Hstmt1->BindStringByName("AName", SName);

//execute the statement to add the new row at the DataSource
Hstmt1->Execute();
```

You can also add a single row at a DataSource without making use of parameters. Just specify the values to insert as part of your SQL statement:

Delphi Code

```
with Hstmt1 do
begin
  //set, prepare and execute an insert statement with embedded values
  SQL:= 'INSERT INTO Students (SNo, SName, SGender) '+
             'VALUES ("9700001", "Van der Merwe", 1)';
  Prepare;
  Execute;
end;
```

C++ Code

```
//set, prepare and execute an insert statement with embedded values
Hstmt1->SQL = "INSERT INTO Students (SNo, SName, SGender) "\
                        "VALUES ('9700001', 'Van der Merwe', 1)";
Hstmt1->Prepare();
Hstmt1->Execute();
```

Update Example

Updating rows at a DataSource is just as easy as inserting rows. The only difference is that usually you have to specify which rows must be updated by way of a "WHERE" clause in the SQL statement:

Delphi Code

```
var
  SNo, SName: String;
  SGender: Byte;
```

```
begin
 with Hstmt1 do
 begin
   //set and prepare an update statement with a where clause to identify the row(s) to be updated
   SQL:= 'UPDATE Students SET SName = ?, SGender = ? WHERE SNo = ?';
   Prepare;

   //set the value of the parameter(s) which will identify the row(s) to be updated
   SNo:= '9700001';

   //set the new values that the row(s) identified must be updated to
   SName:= 'Hamilton';
   SGender:= 0;

   //bind the parameters
   BindString(1, SName);
   BindByte(2, SGender);
   BindString(3, SNo);

   //execute the statement to update the row(s)
   Execute;
 end;
end;
```

C++ Code

```
AnsiString SNo, SName;
Byte SGender;

//set and prepare an update statement with a where clause to identify the row(s) to be updated
Hstmt1->SQL = "UPDATE Students SET SName = ?, SGender = ? WHERE SNo = ?";
Hstmt1->Prepare();

//set the value of the parameter(s) which will identify the row(s) to be updated
SNo = "9700001";

//set the new values that the row(s) identified must be updated to
SName = "Hamilton";
SGender = 0;

//bind the parameters
Hstmt1->BindString(1, SName);
Hstmt1->BindByte(2, SGender);
Hstmt1->BindString(3, SNo);

//execute the statement to update the row(s)
Hstmt1->Execute();
```

If you only want to update a single row, then make sure that the columns in the where clause uniquely identifies the row to be updated. This is usually done by placing all the columns that make up the primary key of a table in the where clause, as in the example above.

You can also update multiple rows by setting, preparing and binding the update statement once, for example say we incorrectly entered the gender of the first 100 students in 1997:

Delphi Code

```
var
  i: Integer;
  SNo: NullString;
  SGender: Byte;
begin
  with Hstmt1 do
  begin
    //set, prepare and bind an update statement once
    SQL:= 'UPDATE Students SET SGender = :AGender WHERE SNo = :ANo';
    Prepare;
    BindNullStringByName('ANo', SNo);
    BindByteByName('AGender', SGender);

    for i:= 1 to 100 do
    begin
      //set the student to update
      StrPCopy(SNo, '97'+IntToStr(i));

      //set the value to be updated
      SGender:= (SGender+1) mod 2;

      //update the gender of a student
      Execute;
    end;
  end;
end;
```

C++ Code

```
int i;
NullString SNo;
Byte SGender;
AnsiString s;

//set, prepare and bind an update statement once
Hstmt1->SQL = "UPDATE Students SET SGender = :AGender WHERE SNo = :ANo";
Hstmt1->Prepare();
Hstmt1->BindNullStringByName("ANo", SNo);
Hstmt1->BindByteByName("AGender", SGender);

for (i=1; i<=100; i++) {
  //set the student to update
  s = "97"+IntToStr(i);
  StrPCopy(SNo, s.c_str());

  //set the value to be updated
  SGender = (Byte)((SGender+1) % 2);

  //update the gender of a student
  Hstmt1->Execute();
}
```

Of course multiple students can also be updated by a single prepare and execute:

Delphi Code

```
with Hstmt1 do
begin
  //perform the same update operation as in the previous example using the ODBC MOD function
  SQL:= 'UPDATE Students SET SGender = {fn MOD(SGender+1, 2)} '+
              'WHERE SNo >= "971" AND SNo <= "97100"';
  Prepare;
  Execute;
end;
```

C++ Code

```
//perform the same update operation as in the previous example using the ODBC MOD function
Hstmt1->SQL = "UPDATE Students SET SGender = {fn MOD(SGender+1, 2)} "\
              "WHERE SNo >= '971' AND SNo <= '97100'";
Hstmt1->Prepare();
Hstmt1->Execute();
```

Delete Example

To delete rows at a DataSource is even simpler than updating rows. Normally a "WHERE" clause is specified to indicate the rows to be deleted at the DataSouce:

Delphi Code

```
with Hstmt1 do
begin
  //set, prepare and execute a delete statement to delete all the students in registered in 1996
  SQL:= 'DELETE FROM Students WHERE SNo LIKE "96%"';
  Prepare;
  Execute;
end;
```

C++ Code

```
//set, prepare and execute a delete statement to delete all the students in registered in 1996
SQL = "DELETE FROM Students WHERE SNo LIKE '96%'";
Hstmt1->Prepare();
Hstmt1->Execute();
```

Another way to delete multiple rows is to set, prepare and bind a delete statement once, and then execute the statement multiple times for different values of the where clause:

Delphi Code

```
var
  i: Integer;
  SNo: NullString;
begin
  with Hstmt1 do
  begin
```

```
    //set, prepare and bind a delete statement
    SQL:= 'DELETE FROM Students WHERE SNo LIKE ?';
    Prepare;
    BindNullString(1, SNo);

    for i:= 80 to 96 do
    begin
      //set the student number mask to indicate the rows to be deleted
      StrPCopy(SNo, IntToStr(i)+'%');

      //execute the statement to delete the students registered in the current year
      Execute;
    end;
  end;
end;
```

C++ Code

```
int i;
NullString SNo;
AnsiString s;

//set, prepare and bind a delete statement
Hstmt1->SQL = "DELETE FROM Students WHERE SNo LIKE ?";
Hstmt1->Prepare();
Hstmt1->BindNullString(1, SNo);

for (i=80; i<=96; i++) {
  //set the student number mask to indicate the rows to be deleted
  s = IntToStr(i)+"%";
  strcpy(SNo, s.c_str());

  //execute the statement to delete the students registered in the current year
  Hstmt1->Execute();
}
```

Simple Select Example

Simple selects are select statements without a "WHERE" clause. Simple select statements are used to select certain columns of all the rows in a table:

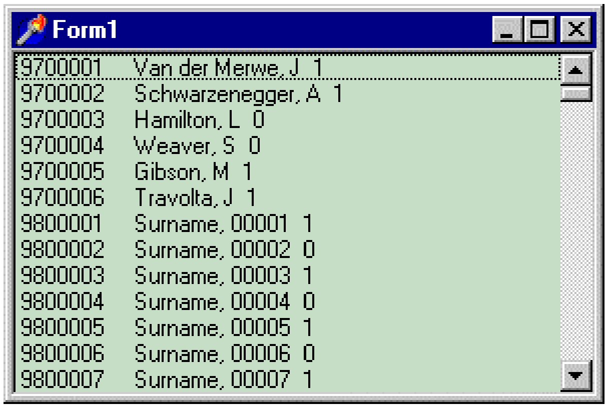
Delphi Code

```
var
  SNo: ^NullString;
  SName: String;
  SGender: Byte;
begin
  with Hstmt1 do
  begin
    //set, prepare and execute a simple select statement
    SQL:= 'SELECT SNo, SName, SGender FROM Students';
    Prepare;
    Execute;
    //point to the student number column - done once for all iterations of the following loop
    SNo:= ColValue(1);

    //fetch each row in the result set one at a time
    while FetchNext do
    begin
      //get the student name and gender columns - done once for each iteration of the loop
      SName:= ColString[2];
      SGender:= ColByte[3];

      //add student information to a ListBox
      ListBox1.Items.Add(SNo^+'  '+SName+'  '+IntToStr(SGender));
    end;
  end;
end;
```

C++ Code

```
NullString* SNo;
AnsiString SName;
Byte SGender;
AnsiString s;

//set, prepare and execute a simple select statement
Hstmt1->SQL = "SELECT SNo, SName, SGender FROM Students";
Hstmt1->Prepare();
Hstmt1->Execute();

//point to the student number column - done once for all iterations of the following loop
SNo = (NullString*)Hstmt1->ColValue(1);

//fetch each row in the result set one at a time
while (Hstmt1->FetchNext()) {
  //get the student name and gender columns - done once for each iteration of the loop
  SName = Hstmt1->ColString[2];
  SGender = Hstmt1->ColByte[3];

  //add student information to a ListBox
  s = "  "+SName+"  "+IntToStr(SGender);
  ListBox1->Items->Add(strcat(*SNo, s.c_str()));
}
```

The list box will look something like this:



```
9700001    Van der Merwe, J  1
9700002    Schwarzenegger, A  1
9700003    Hamilton, L  0
9700004    Weaver, S  0
9700005    Gibson, M  1
9700006    Travolta, J  1
9800001    Surname, 00001  1
9800002    Surname, 00002  0
9800003    Surname, 00003  1
9800004    Surname, 00004  0
9800005    Surname, 00005  1
9800006    Surname, 00006  0
9800007    Surname, 00007  1
```

Another way of referencing the columns returned in the result set of the statement component is by name:

Delphi Code

```
var
  SNo: String;
  SName: String;
  SGender: Byte;
begin
  with Hstmt1 do
  begin
    SQL:= 'SELECT SNo, SName, SGender FROM Students';
    Prepare;
    Execute;

    while FetchNext do
    begin
      //reference the columns in the result set by name using the ColByName methods
      SGender:= ColByteByName['SGender'];
      SNo:= ColStringByName['SNo'];
      SName:= ColStringByName['SName'];

      ListBox1.Items.Add(SNo+'  '+SName+'  '+IntToStr(SGender));
    end;
  end;
end;
```

C++ Code

```
AnsiString SNo;
AnsiString SName;
Byte SGender;

Hstmt1->SQL = "SELECT SNo, SName, SGender FROM Students";
Hstmt1->Prepare();
Hstmt1->Execute();

while (Hstmt1->FetchNext()) {
  //reference the columns in the result set by name using the ColByName methods
  SGender = Hstmt1->ColByteByName["SGender"];
  SNo = Hstmt1->ColStringByName["SNo"];
  SName = Hstmt1->ColStringByName["SName"];

  ListBox1->Items->Add(SNo+"  "+SName+"  "+IntToStr(SGender));
}
```

Complex Select Example

Normally when you do a select from a database, you specify a "WHERE" clause to indicate the rows to be included in the result set of the select statement. Tables at a DataSource can become very large and it's therefore not a good idea to select all the rows in the table to the front-end. We can turn the last piece of code in the simple select example into an complex select example by adding a where clause:

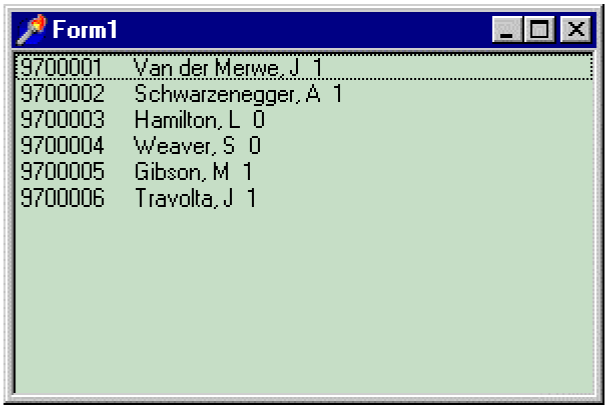
Delphi Code

```
var
  SNo: String;
  SName: String;
  SGender: Byte;
begin
  with Hstmt1 do
  begin
    //set, prepare and execute a select statement with a where clause to limited the rows returned
    SQL:= 'SELECT SNo, SName, SGender FROM Students WHERE SNo LIKE ''97%''';
    Prepare;
    Execute;

    while FetchNext do
    begin
      SGender:= ColByteByName['SGender'];
      SNo:= ColStringByName['SNo'];
      SName:= ColStringByName['SName'];

      ListBox1.Items.Add(SNo+'  '+SName+'  '+IntToStr(SGender));
    end;
  end;
end;
```

C++ Code

```
AnsiString SNo;
AnsiString SName;
Byte SGender;

Hstmt1->SQL = "SELECT SNo, SName, SGender FROM Students  WHERE SNo LIKE '97%'";
Hstmt1->Prepare();
Hstmt1->Execute();

while (Hstmt1->FetchNext()) {
  //reference the columns in the result set by name using the ColByName methods
  SGender = Hstmt1->ColByteByName["SGender"];
  SNo = Hstmt1->ColStringByName["SNo"];
  SName = Hstmt1->ColStringByName["SName"];
  ListBox1->Items->Add(SNo+"  "+SName+"  "+IntToStr(SGender));
}
```

The list box will look something like this:



```
9700001   Van der Merwe, J  1
9700002   Schwarzenegger, A  1
9700003   Hamilton, L  0
9700004   Weaver, S  0
9700005   Gibson, M  1
9700006   Travolta, J  1
```

We can now set, prepare and bind a select statement once, and by just changing the values of the where clause we can select a new result set from a DataSource on each execute:

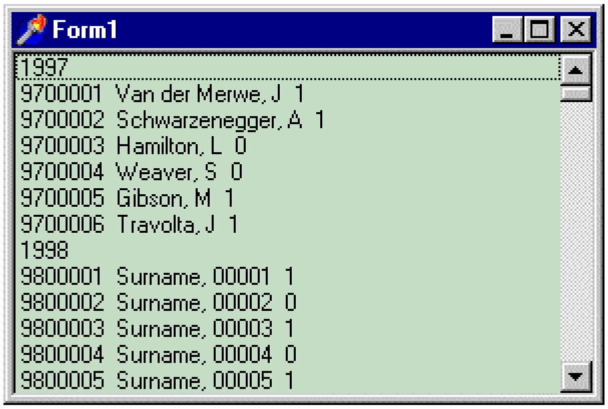
Delphi Code

```
var
  SNo: NullString;
begin
  with Hstmt1 do
  begin
    //set, prepare and bind a select statement
    SQL:= 'SELECT SNo, SName, SGender FROM Students WHERE SNo LIKE ?';
    Prepare;
    BindNullString(1, SNo);

    //indicate and select the first result set
    SNo:= '97%';
    Execute;

    ListBox1.Items.Add('1997');
    while FetchNext do
      ListBox1.Items.Add(ColStringByName['SNo']+'  '+
                                        ColStringByName['SName']+'  '+
                                        ColStringByName['SGender']);

    //indicate and select the second result set
    SNo:= '98%';
    Execute;

    ListBox1.Items.Add('1998');
    while FetchNext do
      ListBox1.Items.Add(ColStringByName['SNo']+'  '+
                                        ColStringByName['SName']+'  '+
                                        ColStringByName['SGender']);
  end;
end;
```

C++ Code

```
NullString SNo;

//set, prepare and bind a select statement
Hstmt1->SQL = "SELECT SNo, SName, SGender FROM Students WHERE SNo LIKE ?";
Hstmt1->Prepare();
Hstmt1->BindNullString(1, SNo);

//indicate and select the first result set
strcpy(SNo, "97%");
Hstmt1->Execute();

ListBox1->Items->Add("1997");
while (Hstmt1->FetchNext()) {
  ListBox1->Items->Add(Hstmt1->ColStringByName["SNo"]+"  "+
                                              Hstmt1->ColStringByName["SName"]+"  "+
                                              Hstmt1->ColStringByName["SGender"]);
}

//indicate and select the second result set
strcpy(SNo, "98%");
Hstmt1->Execute();

ListBox1->Items->Add("1998");
while (Hstmt1->FetchNext()) {
  ListBox1->Items->Add(Hstmt1->ColStringByName["SNo"]+"  "+
                                              Hstmt1->ColStringByName["SName"]+"  "+
                                              Hstmt1->ColStringByName["SGender"]);
}
```

The list box will look something like this:



```
1997
9700001  Van der Merwe, J  1
9700002  Schwarzenegger, A  1
9700003  Hamilton, L  0
9700004  Weaver, S  0
9700005  Gibson, M  1
9700006  Travolta, J  1
1998
9800001  Surname, 00001  1
9800002  Surname, 00002  0
9800003  Surname, 00003  1
9800004  Surname, 00004  0
9800005  Surname, 00005  1
```

BLOb Handling

Storing and retrieving blob fields at a DataSource is just as easy as manipulating any other field at the DataSource. Fields in a database which can hold information larger than 256 bytes are generally BLObs (Binary Large Objects). You usually get two types of blobs:

Text blobs	Consists of a string of readable characters which can't be fitted into a normal 256 character-limited string field.
Binary blobs	Consists of a stream of unreadable characters which is to be stored in binary format in the database.

The blob examples below all make use of the Students table defined in the SQL Operations topic.

Examples:

Example 1	Inserting blobs into the table.
Example 2	Updating blobs in the table.
Example 3	Selecting blobs from the table.

Blob Insert Example

To insert a blob into a database is as simple as doing an insert statement. In this first piece of code we will insert a binary blob in the form of a memory steam:

Delphi Code

```
var
  SNo, SName: String;
  SPic: TMemoryStream;
begin
  //create a memory stream object
  SPic:= TMemoryStream.Create;

  with Hstmt1 do
  begin
    //set and prepare an insert statement with a blob column, SPic
    SQL:= 'INSERT INTO Students (SNo, SName, SGender, SPic) VALUES (?, ?, 1, ?)';
    Prepare;

    //assign values to the parameters and read some data into the memory stream
    SNo:= '9700001';
    SName:= 'Van der Merwe';
    SPic.LoadFromFile('STUDENT.PIC');

    //use the BindBinary method to bind the blob
    BindString(1, SNo);
    BindString(2, SName);
    BindBinary(3, SPic);

    //execute the statement to insert the blob at the DataSource
    Execute;
  end;

  //the memory stream can be destroyed after the execute
  SPic.Free;
end;
```

C++ Code

```
AnsiString SNo, SName;
TMemoryStream* SPic;

//create a memory stream object
SPic = new TMemoryStream();

//set and prepare an insert statement with a blob column, SPic
Hstmt1->SQL = "INSERT INTO Students (SNo, SName, SGender, SPic) VALUES (?, ?, 1, ?)";
Hstmt1->Prepare();

//assign values to the parameters and read some data into the memory stream
SNo = "9700001";
SName = "Van der Merwe";
SPic->LoadFromFile("STUDENT.PIC");

//use the BindBinary method to bind the blob
Hstmt1->BindString(1, SNo);
Hstmt1->BindString(2, SName);
Hstmt1->BindBinary(3, SPic);

//execute the statement to insert the blob at the DataSource
Hstmt1->Execute();

//the memory stream can be destroyed after the execute
delete SPic;
```

To bind a text blob, the same procedure as above is followed, with the exception that the BindText method instead of the BindBinary method is used to bind the text blob into the SQL statement. So what happens when you make a call to the BindBinary or BindText method? Two properties of the memory stream object is used in the process:

Memory	Used to retrieve and bind a pointer to the data using one of the base bind methods BindParam or BindParamCore.
Size	Used to set the size of the bound blob using the ParamSize property.

For example, BindBinary(3, SPic) is similar to the following two lines of code:

Delphi Code

```
Hstmt1.BindParam(3, SQL_C_BINARY, SPic.Memory);
Hstmt1.ParamSize[3]:= SPic.Size;
```

C++ Code

```
Hstmt1->BindParam(3, SQL_C_BINARY, SPic->Memory);
Hstmt1->ParamSize[3] = SPic->Size;
```

So you can see it's pretty easy to bind blobs, even if you don't have the blob in a memory stream object format, as the following example illustrates:

Delphi Code

```
var
  SNo, SName: String;
  SPic: String;
begin
  with Hstmt1 do
  begin
    SQL:= 'INSERT INTO Students (SNo, SName, SGender, SPic) VALUES (?, ?, 1, ?)';
    Prepare;

    //assign a large memo string to the SPic parameter
    SNo:= '9700001';
    SName:= 'Van der Merwe';
    SPic:= Memo1.Text;

    //bind a pointer to the first character in the long string using the BindParam method
    BindString(1, SNo);
    BindString(2, SName);
    BindParam(3, SQL_C_CHAR, PChar(SPic));

    //don't forget to set the size of the blob when you use the BindParam or BindParamCore method
    ParamSize[3]:= Length(SPic);

    //execute the statement to insert the blob into the database
    Execute;
  end;
```

C++ Code

```
AnsiString SNo, SName;
AnsiString SPic;

Hstmt1->SQL = "INSERT INTO Students (SNo, SName, SGender, SPic) VALUES (?, ?, 1, ?)";
Hstmt1->Prepare();

//assign a large memo string to the SPic parameter
SNo = "9700001";
SName = "Van der Merwe";
SPic = Memo1->Text;

//bind a pointer to the first character in the long string using the BindParam method
Hstmt1->BindString(1, SNo);
Hstmt1->BindString(2, SName);
Hstmt1->BindParam(3, SQL_C_CHAR, SPic.c_str());

//don't forget to set the size of the blob when you use the BindParam or BindParamCore method
Hstmt1->ParamSize[3] = SPic.Length();

//execute the statement to insert the blob into the database
Hstmt1->Execute();
```

Blob Update Example

Updating a blob is just as easy as inserting a blob. Here's a quick example to illustrate this:

Delphi Code

```
var
  SNo: String;
  SPic: TMemoryStream;
begin
  SPic:= TMemoryStream.Create;

  with Hstmt1 do
  begin
    //set and prepare an update statement with a blob field
    SQL:= 'UPDATE Students SET SPic = :APic WHERE SNo = :ANo';
    Prepare;

    //assign values to the parameters and load data into the memory stream
    SNo:= '9700001';
    SPic.LoadFromFile('STUDENT.PIC');

    //bind the memory stream using the BindBinary method
    BindStringByName('ANo', SNo);
    BindBinaryByName('APic', SPic);

    //update the blob in the database
    Execute;
  end;

  SPic.Free;
end;
```

C++ Code

```
AnsiString SNo;
TMemoryStream* SPic;

SPic = new TMemoryStream();

//set and prepare an update statement with a blob field
Hstmt1->SQL = "UPDATE Students SET SPic = :APic WHERE SNo = :ANo";
Hstmt1->Prepare();

//assign values to the parameters and load data into the memory stream
SNo = "9700001";
SPic->LoadFromFile("STUDENT.PIC");

//bind the memory stream using the BindBinary method
Hstmt1->BindStringByName("ANo", SNo);
Hstmt1->BindBinaryByName("APic", SPic);

//update the blob in the database
Hstmt1->Execute();

delete SPic;
```

Blob Select Example

Selecting a blob from a database is even easier than inserting or updating a blob:

Delphi Code

```
var
  SNo: String;
begin
  with Hstmt1 do
  begin
    //set, prepare, bind and execute a select statement which includes a blob column, SPic
    SQL:= 'SELECT SNo, SName, SPic FROM Students WHERE SNo = ?';
    Prepare;
    SNo:= '9700001';
    BindString(1, SNo);
    Execute;

    if FetchNext then
    begin
      //access the blob through the ColMemory property - it's as simple as that
      ColMemoryByName['SPic'].SaveToFile('STUDENT.PIC');

      ShowMessage('Student''s picture saved: '+ColStringByName['SNo']);
    end;
  end;
end;
```

C++ Code

```
AnsiString SNo;

//set, prepare, bind and execute a select statement which includes a blob column, SPic
Hstmt1->SQL = "SELECT SNo, SName, SPic FROM Students WHERE SNo = ?";
Hstmt1->Prepare();
SNo = "9700001";
Hstmt1->BindString(1, SNo);
Hstmt1->Execute();

if (Hstmt1->FetchNext()) {
  //access the blob through the ColMemory property - it's as simple as that
  Hstmt1->ColMemoryByName["SPic"]->SaveToFile("STUDENT.PIC");

  ShowMessage("Student's picture saved: "+Hstmt1->ColStringByName["SNo"]);
}
```

Note:

Some ODBC drivers require the blob fields in a select statement to be the last columns in the select column list, for example the blob column SPic is the last column in the select column list "SNo, SName, SPic". This has to do with the way that blobs are stored in a database, so for maximum interoperability it is recommended to always put the blob columns last in a list of columns. It's also a good idea to define the blob columns in the SQL statement in the same order they appear in the table.

SQL Generation

There are two distinct ways to insert, update or delete a row at a DataSource:

1. Specifying an insert, update or delete SQL statement and executing it to insert, update or delete a row.
2. Specifying and executing a select SQL statement and doing a positional insert, update or delete of a row in the result set.

While in the first case you always have to set up your own SQL statement, in the second case the desired "positional operation" can be performed either using no SQL, automatically generated SQL, or manually constructed SQL. These three options form the basis of ODBCExpress' SQL Generation functionality for positional operations.

When doing a positional insert, update or delete, the success of the operation depends on if the ODBC driver supports the operation and if the result set is modifiable. These two issues can be solved by using the correct level of SQL Generation (provided of course that the user has the appropriate rights to modify the data in the result set). Four levels of SQL Generation are provided:

Level 1

SQL-less operations using ODBC functions to do the inserts, updates, deletes and also refreshes of rows in the result set. This requires the least effort and control, but uses level 2 ODBC functionality to perform the operations. The level 2 ODBC functionality required to perform these operations aren't supported by all ODBC drivers.

Level 2

Generated SQL using result set cursor positions to perform updates and deletes. Complete SQL statements are generated to perform each operation, making use of the result set cursor to determine the row affected by the operation. It might be necessary to supply the name of the table which will be affected, but only if the ODBC driver is not able to detect it automatically.

Level 3

Generated SQL using indicated primary keys when necessary to perform inserts, updates and deletes. Complete SQL statements are also generated to perform each operation, but in this case the primary key columns, as set using the ColPrimary or PrimaryCols properties of the statement component, is used to determine the rows affected by the operation. Indicating the primary key columns is of course not necessary when doing deletes.

Level 4

Manual SQL construction using events to do inserts, updates, deletes and also refreshes of rows in the result set. This is done by specifying your own SQL statements to perform the operations using the OnInsert, OnUpdate, OnDelete and OnRefresh events of the statement component.

Levels 2, 3 and 4 can be compared to the functionality performed by the Delphi and C++Builder TUpdateSQL component.

Levels 3 and 4 can be used to modify potentially non-modifiable result sets. A result set is non-modifiable when the concurrency type of the result set is read-only, or if the result set was returned by a query which makes it non-modifiable, such as a join. Using these two levels will allow you to bypass the non-modifiable state of the result set, allowing the user to do positional inserts, updates and deletes, provided of course that the user has the appropriate rights to modify the data at the DataSource.

Positional inserts, updates, deletes and refreshes are performed usng the DoInsert, DoUpdate, DoDelete and DoRefresh methods of the statement component respectively.

The following shows how to instruct the statement component to use the desired level of SQL Generation:

1. Level 4 generation can override any of the other levels by implementing the appropriate operation inside the OnInsert, OnUpdate, OnDelete or OnRefresh event, and then returning False from the event to avoid any of the first 3 levels used. These events can therefore also be used to confirm an insert, update, delete or refresh operation, by returning the appropriate boolean value from the event.

2. When level 4 generation isn't done, by default level 1, 2 or 3 generation is done, depending on the first level supported by the ODBC driver used. The statement component will detect which of these 3 levels can be performed by the ODBC driver. If level 1 generation is not supported by the driver, level 2 generation is attempted and if level 2 generation is also not supported by the driver, then level 3 generation is done. Level 3 generation will work for any ODBC driver, no matter what the API conformance level of the ODBC driver is.

3. Level 1 generation can be completely bypassed by setting the SkipByPosition property of the statement component to True. Then only level 2 or level 3 generation will be attempted when level 4 generation isn't done.

4. Level 2 generation can be completely bypassed by setting the SkipByCursor property of the statement component to True. Then only level 1 or level 3 generation will be attempted when level 4 generation isn't done.

5. Level 1 and level 2 generation can be completely bypassed by setting both the SkipByPosition and SkipByCursor properties of the statement component to True. Then only level 3 generation will be attempted when level 4 generation isn't done.

Examples:

Example 1	Level 1 SQL Generation
Example 2	Level 2 SQL Generation
Example 3	Level 3 SQL Generation
Example 4	Level 4 SQL Generation

Level 1 Example

Delphi Code

```
//suppose we generate the following result set

with Hstmt1 do
begin
  SQL:= 'SELECT SNo, SName, SGender, SBirth FROM Students';
  Prepare;
  Execute;
end;

//fetch some rows

//then to update the current row using Level 1 SQL Generation

with Hstmt1 do
begin
  ColStringByName['SName']:= 'Hamilton, L';
  DoUpdate;
end;
```

C++ Code

```
//suppose we generate the following result set

Hstmt1->SQL = "SELECT SNo, SName, SGender, SBirth FROM Students";
Hstmt1->Prepare();
Hstmt1->Execute();

//fetch some rows

//then to update the current row using Level 1 SQL Generation

Hstmt1->ColStringByName["SName"] = "Hamilton, L";
Hstmt1->DoUpdate();
```

Note:

If level 1 SQL Generation can't be done, it will automatically try level 2 and then level 3 SQL Generation in this case.

Level 2 Example

Delphi Code

```
//suppose we generate the following result set

with Hstmt1 do
begin
  SQL:= 'SELECT SNo, SName, SGender, SBirth FROM Students';
  Prepare;
  Execute;
end;

//fetch some rows

//then to update the current row using Level 2 SQL Generation

with Hstmt1 do
begin
  ColStringByName['SName']:= 'Hamilton, L';
  //skip Level 1 SQL Generation
  SkipByPosition:= True;

  //some ODBC Drivers aren't able to detect the target table
  TargetTable:= 'Students';

  DoUpdate;
end;
```

C++ Code

```
//suppose we generate the following result set

Hstmt1->SQL = "SELECT SNo, SName, SGender, SBirth FROM Students";
Hstmt1->Prepare();
Hstmt1->Execute();

//fetch some rows

//then to update the current row using Level 2 SQL Generation

Hstmt1->ColStringByName["SName"] = "Hamilton, L";

//skip Level 1 SQL Generation
Hstmt1->SkipByPosition = True;

//some ODBC Drivers aren't able to detect the target table
TargetTable = "Students";

Hstmt1->DoUpdate();
```

Note:

If level 2 SQL Generation can't be done, it will automatically try level 3 SQL Generation in this case.

Level 3 Example

Delphi Code

```
//suppose we generate the following result set

with Hstmt1 do
begin
  SQL:= 'SELECT SNo, SName, SGender, SBirth FROM Students';
  Prepare;
  Execute;
end;

//fetch some rows

//then to update the current row using Level 2 SQL Generation
```

```
with Hstmt1 do
begin
  ColStringByName['SName']:= 'Hamilton, L';

  //skip Level 1 and Level 2 SQL Generation
  SkipByPosition:= True;
  SkipByCursor:= True;

  //some ODBC Drivers aren't able to detect the target table
  TargetTable:= 'Students';

  //indicate the columns that make up the primary key
  PrimaryCols(['SNo']);

  DoUpdate;
end;
```

C++ Code

```
//suppose we generate the following result set

Hstmt1->SQL = "SELECT SNo, SName, SGender, SBirth FROM Students";
Hstmt1->Prepare();
Hstmt1->Execute();

//fetch some rows

//then to update the current row using Level 2 SQL Generation

Hstmt1->ColStringByName["SName"] = "Hamilton, L";

//skip Level 1 and Level 2 SQL Generation
Hstmt1->SkipByPosition = True;
Hstmt1->SkipByCursor = True;

//some ODBC Drivers aren't able to detect the target table
Hstmt1->TargetTable = "Students";

//indicate the columns that make up the primary key
Hstmt1->PrimaryCols(OPENARRAY(AnsiString, ("SNo")));

Hstmt1->DoUpdate();
```

Level 4 Example

Delphi Code

```
//suppose we generate the following result set

with Hstmt1 do
begin
  SQL:= 'SELECT SNo, SName, SGender, SBirth FROM Students';
  Prepare;
  Execute;
end;

//fetch some rows

//implement the OnUpdate event of Hstmt1

function TForm1.Hstmt1Update(Sender: TObject; DefaultMsg: String): Boolean;
var
  SNo, SName: String;
  SGender: Byte;
  SBirth: TTimeStamp;
begin
  //get user confirmation of update
  Result:= MessageDLG(DefaultMsg, mtConfirmation, [mbOK, mbCancel], 0) = mrOK;

  //perform Level 4 SQL Generation
  if Result then
  begin
    //perform a manual update using a different statement component
    with Hstmt2 do
    begin
      //retrieve column values
      SNo:= Hstmt1.ColStringByName['SNo'];
      SName:= Hstmt1.ColStringByName['SName'];
      SGender:= Hstmt1.ColByteByName['SGender'];
      SBirth:= Hstmt1.ColTimeStampByName['SBirth'];

      //set and prepare SQL statement
      SQL:= 'UPDATE Students SET SName = ?, SGender = ?, SBirth = ? WHERE SNo = ?';
      Prepare;

      //bind column values
      BindString(1, SName);
      BindByte(2, SGender);
      BindTimeStamp(3, SBirth);
      BindString(4, SNo);

      //update row
      Execute;
    end;

    //skip first 3 Levels of SQL Generation
    Result:= False;
  end;
end;
```

C++ Code

```
//suppose we generate the following result set

Hstmt1->SQL = "SELECT SNo, SName, SGender, SBirth FROM Students";
Hstmt1->Prepare();
Hstmt1->Execute();

//fetch some rows
```

```
//implement the OnUpdate event of Hstmt1

bool __fastcall TForm1::Hstmt1Update(TObject *Sender, AnsiString DefaultMsg)
{
  AnsiString SNo, SName;
  Byte SGender;
  Ocih::TTimeStamp SBirth;
  bool Result;

  //get user confirmation of update
  Result = MessageDlg(DefaultMsg, mtConfirmation, TMsgDlgButtons() << mbOK << mbCancel,
0) == mrOk;

  //perform Level 4 SQL Generation
  if (Result) {
    //perform a manual update using a different statement component

    //retrieve column values
    SNo = Hstmt1->ColStringByName["SNo"];
    SName = Hstmt1->ColStringByName["SName"];
    SGender = Hstmt1->ColByteByName["SGender"];
    SBirth = Hstmt1->ColTimeStampByName["SBirth"];

    //set and prepare SQL statement
    Hstmt2->SQL = "UPDATE Students SET SName = ?, SGender = ?, SBirth = ? WHERE SNo =
?";
    Hstmt2->Prepare();

    //bind column values
    Hstmt2->BindString(1, SName);
    Hstmt2->BindByte(2, SGender);
    Hstmt2->BindTimeStamp(3, SBirth);
    Hstmt2->BindString(4, SNo);

    //update row
    Hstmt2->Execute();

    //skip first 3 Levels of SQL Generation
    Result = False;
  }

  return Result;
}
```

Transactioning

Transactioning in ODBCExpress is done at connection component level. The connection component provides methods to start, end, commit and rollback transactions. By default, the connection component is in auto-commit mode. This means that all operations done at a DataSource using the child statement components of a connection component are automatically committed. Starting transaction mode takes the connection component out of auto-commit mode and starts a new transaction. Performing multiple operations in transaction mode is much faster than performing the operations out of transaction mode, since an auto-commit won't be done after each operation.

A new transaction is started after each call to THdbc.StartTransact, THdbc.Commit and THdbc.RollBack, while a transaction is ended by each call to THdbc.Commit, THdbc.RollBack and THdbc.EndTransact. Note that a transaction only starts at the first statement call made after StartTransact, Commit or RollBack was called. Also, calling Commit, RollBack or EndTransact without preceding it with transactional operations has no effect and won't cause any errors. Always match each StartTransact with an EndTransact, preferably as part of a try-finally statement, otherwise your application will end up in an invalid transaction state when an exception occurs.

The following example illustrates how to do simple transactioning:

Example 1 Simple Transactioning

Because transactioning is done at connection component level, it means that the transactions done on different connection components within the same application are isolated from each other, as well as transactions done from different applications. Care must therefore be taken when using multiple connection components in the same application to avoid locking problems, as illustrated in the following example:

Example 2 Transaction Locking

However, different statement components on the same connection component aren't isolated from each other when in transaction mode, which means you can make use of multiple statement component within a single transaction to perform the operations you require at a DataSource, as the following example illustrates:

Example 3 Multiple Statement Transactioning

To make transactioning a bit easier, here's a simple transactioning strategy that can be followed in your application:

Only make use of transactioning on one main connection component within your application to avoid locking problems between connections in your application.
Other connection components can be used to perform labour intensive operations, such as large queries on different threads in your application.
For your main connection component, have two main statement components. The first statement component can have a forward-only cursor to perform operations such as inserts, updates and deletes at a DataSource. The second statement component can have a forward-only or scrollable cursor, probably read-only, to perform select operations from a DataSource.
Other statement components with scrollable cursors on the connection component can be used to control the data-aware Visual Components in your application.

Simple Transactioning Example

Delphi Code

```
var
  i: Integer;
  SNo, SName: NullString;
  SGender: Byte;
begin
  //start manual commit mode
```

```
 Hdbc1.StartTransact;

 try

  with Hstmt1 do
  begin
   //set, prepare and bind an insert statement
   SQL:= 'INSERT INTO Students (SNo, SName, SGender) VALUES (?, ?, ?)';
   Prepare;
   BindNullString(1, SNo);
   BindNullString(2, SName);
   BindByte(3, SGender);

   //execute multiple statements
   for i:= 1 to 100 do
   begin
    StrPCopy(SNo, '97'+IntToStr(i));
    StrPCopy(SName, 'Student'+IntToStr(i));
    SGender:= i mod 2;
    Execute;
   end;
  end;

 //rollback transaction when an exception occurs
 except
  on EODBC do
   Hdbc1.Rollback;
 end;

 //commit transaction
 Hdbc1.Commit;

 //end manual commit mode
 Hdbc1.EndTransact;
end;
```

C++ Code

```
int i;
NullString SNo, SName;
Byte SGender;
AnsiString s;

//start manual commit mode
Hdbc1->StartTransact();

try {

 //set, prepare and bind an insert statement
 Hstmt1->SQL = "INSERT INTO Students (SNo, SName, SGender) VALUES (?, ?, ?)";
 Hstmt1->Prepare();
 Hstmt1->BindNullString(1, SNo);
 Hstmt1->BindNullString(2, SName);
 Hstmt1->BindByte(3, SGender);

 //execute multiple statements
```

```
  for (i = 1; i <= 100; i++) {
    s = "97"+IntToStr(i);
    strcpy(SNo, s.c_str());
    s = "Student"+IntToStr(i);
    strcpy(SName, s.c_str());
    SGender = (Byte)(i % 2);
    Hstmt1->Execute();
  }

}
//rollback transaction when an exception occurs
catch (EODBC *E) {
  Hdbc1->Rollback();
}

//commit transaction
Hdbc1->Commit();

//end manual commit mode
Hdbc1->EndTransact();
```

Transaction Locking Example

Delphi Code

```
var
  SNo1, SName1: NullString;
  SGender1: Byte;
  SNo2, SName2: NullString;
  SGender2: Byte;
begin
  //start transactions on two connection classes
  Hdbc1.StartTransact;
  Hdbc2.StartTransact;

  //set, prepare and bind a statement using a child statement class of Hdbc1
  with Hstmt1 do
  begin
    SQL:= 'INSERT INTO Students (SNo, SName, SGender) VALUES (?, ?, ?)';
    Prepare;
    BindNullString(1, SNo1);
    BindNullString(2, SName1);
    BindByte(3, SGender1);
  end;

  //set, prepare and bind a statement using a child statement class of Hdbc2
  with Hstmt2 do
  begin
    SQL:= 'INSERT INTO Students (SNo, SName, SGender) VALUES (?, ?, ?)';
    Prepare;
    BindNullString(1, SNo2);
    BindNullString(2, SName2);
    BindByte(3, SGender2);
  end;
  //execute one statement without committing
  Hstmt1.Execute;

  //the application will hang on this execute, since it waits for the previous execute to commit
  Hstmt2.Execute;

  Hdbc1.EndTransact;
  Hdbc2.EndTransact;
end;
```

C++ Code

```
NullString SNo1, SName1;
Byte SGender1;
NullString SNo2, SName2;
Byte SGender2;

//start transactions on two connection classes
Hdbc1->StartTransact();
Hdbc2->StartTransact();

//set, prepare and bind a statement using a child statement class of Hdbc1
Hstmt1->SQL = "INSERT INTO Students (SNo, SName, SGender) VALUES (?, ?, ?)";
Hstmt1->Prepare();
Hstmt1->BindNullString(1, SNo1);
Hstmt1->BindNullString(2, SName1);
Hstmt1->BindByte(3, SGender1);

//set, prepare and bind a statement using a child statement class of Hdbc2
Hstmt2->SQL = "INSERT INTO Students (SNo, SName, SGender) VALUES (?, ?, ?)";
Hstmt2->Prepare();
Hstmt2->BindNullString(1, SNo2);
Hstmt2->BindNullString(2, SName2);
Hstmt2->BindByte(3, SGender2);

//execute one statement without committing
Hstmt1->Execute();

//the application will hang on this execute, since it waits for the previous execute to commit
Hstmt2->Execute();

Hdbc1->EndTransact();
Hdbc2->EndTransact();
```

Multiple Statement Transactioning Example

Delphi Code

```
var
  SNo1, SName1: NullString;
  SGender1: Byte;
  SNo2, SName2: NullString;
  SGender2: Byte;
```

```
begin
  //start transaction mode
  Hdbc1.StartTransact;

  //set, prepare and execute a statement class on Hdbc1
  with Hstmt1 do
  begin
    SQL:= 'INSERT INTO Students (SNo, SName, SGender) VALUES (?, ?, ?)';
    Prepare;
    BindNullString(1, SNo1);
    BindNullString(2, SName1);
    BindByte(3, SGender1);
  end;

  //set, prepare and execute another statement class on Hdbc1
  with Hstmt2 do
  begin
    SQL:= 'INSERT INTO Students (SNo, SName, SGender) VALUES (?, ?, ?)';
    Prepare;
    BindNullString(1, SNo2);
    BindNullString(2, SName2);
    BindByte(3, SGender2);
  end;

  //execute both statements
  Hstmt1.Execute;
  Hstmt2.Execute;

  //commit the transaction and end transaction mode
  Hdbc1.EndTransact;
end;
```

C++ Code

```
NullString SNo1, SName1;
Byte SGender1;
NullString SNo2, SName2;
Byte SGender2;

//start transaction mode
Hdbc1->StartTransact();

//set, prepare and execute a statement class on Hdbc1
Hstmt1->SQL = "INSERT INTO Students (SNo, SName, SGender) VALUES (?, ?, ?)";
Hstmt1->Prepare();
Hstmt1->BindNullString(1, SNo1);
Hstmt1->BindNullString(2, SName1);
Hstmt1->BindByte(3, SGender1);

//set, prepare and execute another statement class on Hdbc1
Hstmt2->SQL = "INSERT INTO Students (SNo, SName, SGender) VALUES (?, ?, ?)";
Hstmt2->Prepare();
Hstmt2->BindNullString(1, SNo2);
Hstmt2->BindNullString(2, SName2);
Hstmt2->BindByte(3, SGender2);
```

```
//execute both statements
Hstmt1->Execute();
Hstmt2->Execute();

//commit the transaction and end transaction mode
Hdbc1->EndTransact();
```

Control Population

Data-aware controls can be populated using the TOEDataSet and descendant components. The TOEDataSet is a descendant of the virtual TDataSet to enable you to use your Delphi or C++Builder and other compliant third-party data-aware controls (such as the TOEIntFloat and TOEDateTime), with ODBCExpress result sets.

To populate Delphi or C++Builder and other compliant third-party data-aware controls via the TOEDataSet, see the Delphi or C++Builder documentation on populating data-aware controls via TDataSet components.

Writing Data-Aware Controls

You get two types of data-aware visual controls:

Single-row Controls	Display only one row (and mostly one column) of data from a DataSource at a time.
Multi-row Controls	Display multiple rows (and usually multiple columns) of data from a DataSource at a time.

To write Delphi or C++Builder single-row and multi-row data-aware controls, which will work with any TDataSet descendant including the TOEDataSet and descendant components, see the Delphi or C++Builder documentation on writing data-aware controls.

Linking ODBC Libraries

ODBCExpress statically links to the ODBC DLLs. However if you have the source code of ODBCExpress, the option is provided to dynamically link to the ODBC DLLs. By defining the compiler directive "ODBCDYN" in the "conditional defines" section of your project options, you can re-compile the ODBCExpress code to dynamically link to the ODBC DLLs. The creation of the global ODBC environment component GlobalHenv is also deferred to avoid the ODBC DLLs being loaded before they are needed.

ODBC Types

The arguments of the ODBC Functions are all declared using ODBC Types. ODBC Types are normally used to distinguish database variables from other variables in a program. However, each ODBC Type corresponds to exactly one Delphi or C++ Type, and can be used interchangably.

Delphi	**C++**

Data Types

The following data types are declared in the unit OCIH:

```
  SQLCHAR = Byte;                                   Byte SQLCHAR;
  SQLSCHAR = Shortint;                              Shortint SQLSCHAR;
  SQLSMALLINT = Smallint;                           short SQLSMALLINT;
  SQLUSMALLINT = Word;                              Word SQLUSMALLINT;
  SQLINTEGER = Integer;                             int SQLINTEGER;
  SQLUINTEGER = Cardinal;                           Cardinal SQLUINTEGER;
..SQLREAL = Single;                                 float SQLREAL;
..SQLDOUBLE = Double;                               double SQLDOUBLE;
  SQLPOINTER = Pointer;                             void* SQLPOINTER;
  SQLBOOL = Boolean;                                bool SQLBOOL;
  SQLBIGINT = Int64                                 __int64 SQLBIGINT {
  SQLUBIGINT = Int64                                __int64 SQLUBIGINT {
  SQLRETURN = SQLSMALLINT;                          SQLSMALLINT SQLRETURN;
  SQLHANDLE = Pointer;                              void* SQLHANDLE;
  SQLHENV = SQLHANDLE;                              SQLHANDLE SQLHENV;
  SQLHDBC = SQLHANDLE;                              SQLHANDLE SQLHDBC;
  SQLHSTMT = SQLHANDLE;                             SQLHANDLE SQLHSTMT;
  SQLHDESC = SQLHANDLE;                             SQLHANDLE SQLHDESC;
  SQLHWND = HWND;                                   HWND SQLHWND;
```

Pointer Types

The following data types are declared in the unit OCIH:

```
  SQLCHARPtr = ^SQLCHAR;                            SQLCHAR* SQLCHARPtr;
  SQLSCHARPtr = ^SQLSCHAR;                          SQLSCHAR* SQLSCHARPtr;
  SQLSMALLINTPtr = ^SQLSMALLINT;                    SQLSMALLINT*
SQLSMALLINTPtr;
  SQLUSMALLINTPtr = ^SQLUSMALLINT;                  SQLUSMALLINT*
SQLUSMALLINTPtr;
  SQLINTEGERPtr = ^SQLINTEGER;                      SQLINTEGER*
SQLINTEGERPtr;
  SQLUINTEGERPtr = ^SQLUINTEGER;                    SQLUINTEGER*
SQLUINTEGERPtr;
  SQLREALPtr = ^SQLREAL;                            SQLREAL* SQLREALPtr;
  SQLDOUBLEPtr = ^SQLDOUBLE;                        SQLDOUBLE*
SQLDOUBLEPtr;
  SQLPOINTERPtr = ^SQLPOINTER;                      SQLPOINTER*
SQLPOINTERPtr;
  SQLBOOLPtr = ^SQLBOOL;                            SQLBOOL* SQLBOOLPtr;
  SQLBIGINTPtr = ^SQLBIGINT;                        SQLBIGINT* SQLBIGINTPtr;
  SQLUBIGINTPtr = ^SQLUBIGINT;                      SQLUBIGINT* SQLUBIGINTPtr;
  SQLHANDLEPtr = ^SQLHANDLE;                        SQLHANDLE* SQLHANDLEPtr;
```

Record Types

The following data types are declared in the unit OCIH:

```
  { TDate }                                         //TDate
  TDate = record                                    struct TDate {
```

```
  year: SQLSMALLINT;
  month: SQLUSMALLINT;
  day: SQLUSMALLINT;
end;

{ TTime }
TTime = record
  hour: SQLUSMALLINT;
  minute: SQLUSMALLINT;
  second: SQLUSMALLINT;
end;

{ TTimeStamp }
TTimeStamp = record
  year: SQLSMALLINT;
  month: SQLUSMALLINT;
  day: SQLUSMALLINT;
  hour: SQLUSMALLINT;
  minute: SQLUSMALLINT;
  second: SQLUSMALLINT;
  fraction: SQLUINTEGER;
end;
```

```
  SQLSMALLINT year;
  SQLUSMALLINT month;
  SQLUSMALLINT day;
};

//TTime
struct TTime {
  SQLUSMALLINT hour;
  SQLUSMALLINT minute;
  SQLUSMALLINT second;
};

//TTimeStamp
struct TTimeStamp {
  SQLSMALLINT year;
  SQLUSMALLINT month;
  SQLUSMALLINT day;
  SQLUSMALLINT hour;
  SQLUSMALLINT minute;
  SQLUSMALLINT second;
  SQLUINTEGER fraction;
};
```

Data Types

The following Delphi and C++ Types are used by the ODBCExpress libraries:

Delphi	**C++**

Data Types

The following data types are declared in the unit OCIH:

```
{ NullString }
NullString = array[0..MaxNullString] of Char;
```

```
//NullString
char
```

```
NullString[MaxNullString+1];
```

Pointer Types

The following data types are declared in the unit OCL:

```
NullStringPtr = ^NullString;
StringPtr = ^String;
SinglePtr = ^Single;
DoublePtr = ^Double;
ShortintPtr = ^Shortint;
BytePtr = ^Byte;
SmallintPtr = ^Smallint;
WordPtr = ^Word;
IntegerPtr = ^Integer;
CardinalPtr = ^Cardinal;
LongintPtr = ^Longint;
```

```
NullString* NullStringPtr;
AnsiString* StringPtr;
float* SinglePtr;
double* DoublePtr;
Shortint* ShortintPtr;
Byte* BytePtr;
short* SmallintPtr;
Word* WordPtr;
int* IntegerPtr;
Cardinal* CardinalPtr;
long* LongPtr;
```

```
LongwordPtr = ^Longword;
Int64Ptr = ^Int64;
TDatePtr = ^TDate;
TTimePtr = ^TTime;
TTimeStampPtr = ^TTimeStamp;
```

```
Longword* LongwordPtr;
__int64* Int64Ptr;
TDate* TDatePtr;
TTime* TTimePtr;
TTimeStamp* TTimeStampPtr;
```

Record Types

The following data types are declared in the unit OCL:

```
{ TRowPtr }
TRowPtr = ^TRowRec;
TRowRec = record
  FType: SQLSMALLINT;
  FValue: SQLPOINTER;
  FSize: SQLINTEGERPtr;
  FBlob: Boolean;
end;
```

```
//TRowPtr
TRowRec* TRowPtr;
struct TRowRec {
  SQLSMALLINT FType;
  SQLPOINTER FValue;
  SQLINTEGERPtr FSize;
  bool FBlob;
};
```

ODBC Constants

The ODBC Sql Type and Storage Type constants can be found in the ODBC Constants Map appendix.

Other ODBC constants are declared, where relevant, throughout the help file.

For a complete list of ODBC constants, see the ODBC documentation.

ODBC Documentation

The Microsoft Data Access SDK Documentation is an excellent reference on ODBC. It contains descriptions of all the ODBC functions, the ODBC constants used with these functions, and all other topics related to ODBC.

See the ODBCExpress Web Site for details on obtaining the Microsoft Data Access SDK and other ODBC related materials.

Data Constants

The following Delphi and C++ Constants are used by the ODBCExpress libraries:

Delphi	**C++**

Constants

The following constants are declared in the unit OCIH:

```
MaxNullString = 255;
```

```
MaxNullString 255;
```

EODBCExpress

Hierarchy

Unit OCIH;

Description

The ODBCExpress exception class is used to raise any exception which occurs in the ODBCExpress code, instead of using the base Exception class to do this. The EODBC exception class, which is used to raise exceptions returned by ODBC, is inherited from this exception class.

EODBC

Hierarchy   Properties   Methods   Example

Unit OCL;

Description

The ODBC exception class is used to trap and handle ODBC errors. ODBC errors can occur in the ODBC Driver Manager, in the ODBC driver, or at the database. An ODBC error raised by ODBCExpress can be trapped and handled using a try-except statement in Dephi or a try-catch statement in C++Builder, which tests for the exception class EODBC. All other errors which occur in ODBCExpress will be returned through the parent exception class EODBCExpress.

ODBC drivers sometimes return multiple errors in an error message queue, which can be browsed using the First, Last, Next and Prev methods of this exception class.

EODBC.Detail

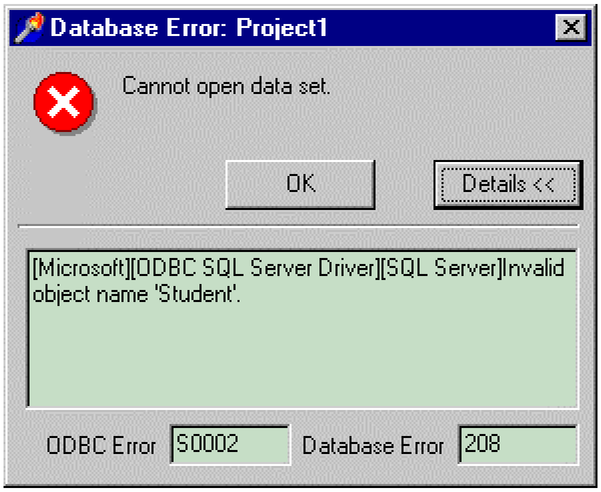
procedure Detail(AMessage: **String**);

void Detail(**AnsiString** AMessage);

Description

Displays an error dialog with a custom error message AMessage for the current ODBC error. The dialog also contains a Detail button, which will display the ODBC error Message, the ODBC error State and the Native database error, as well as Next and Prev buttons to scroll through the error message queue.

Example:



Cannot open data set.

[Microsoft][ODBC SQL Server Driver][SQL Server]Invalid object name 'Student'.

ODBC Error: S0002

Database Error: 208

EODBC.Display

procedure Display;

void Display(**void**);

Description

Displays the ODBC error message of the current ODBC error, as returned by the Message property.

EODBC.First

procedure First;

void First(**void**);

Description

Used to position to the first error message in the queue. This is the initial position.

EODBC.Last

procedure Last;

void Last(**void**);

Description

Used to position to the last error message in the queue.

EODBC.Message

property Message: **String**;

property AnsiString Message;

Description

Returns the ODBC error message of the current ODBC error. The error message usually indicates where the error occured by means of prefixes enclosed in square brackets. An ODBC error can occur in the ODBC Driver Manager, in the ODBC driver, or at the database. This property can also be set to modify the default error message.

Example:

The following error occured at the database:
[Microsoft][ODBC SQL Server Driver][SQL Server] Invalid object name 'Students'.

The following error occured at the ODBC Driver Manager:
[Microsoft][ODBC Driver Manager] Data source name not found and no default driver specified.

EODBC Native

property Native: SQLINTEGER;

property SQLINTEGER Native;

Description

Returns the native database error code of the current ODBC error as an integer value.

EODBC.Next

function Next: **Boolean**;

bool Next(**void**);

Description

Used to position to the next error message in the queue. If there is no more error messages after the current one in the queue, this method will return False.

EODBC.Owner

property Owner: THbase;

property THbase* Owner;

Description

Returns the owner component of the current error. It can be one of the three handle components THenv, THdbc and THstmt. If the owner component is a THdbc component, the owner THenv component is the global variable GlobalHenv (contained in the OCL unit). If the owner component is a THstmt component, the owner THdbc component can be found through the

THstmt.Hdbc property and the owner THenv component is again the global variable GlobalHenv.

EODBC.Prev

function Prev: **Boolean**;

bool Prev(**void**);

Description

Used to position to the previous error message in the queue. If there is no more error messages before the current one in the queue, this method will return False.

EODBC.RetCode

property RetCode: SQLRETURN;

property SQLRETURN RetCode;

Description

Returns the return code of the ODBC function which caused the current exception. It normally has one of the following values:

SQL_ERROR	Indicates that the ODBC function did not successfully execute. An exception is always raised for an error of this type.
SQL_SUCCESS_WITH_INFO	Indicates that the ODBC function successfully executed, but there was some non-serious negative affects. An exception is only raised for an error of this type if GlobalHenv.Error.RaiseSoftErrors is set to True.
SQL_INVALID_HANDLE	Indicates that the handle variable passed to the ODBC function is not a valid ODBC handle. An exception is always raised for an error of this type.

EODBC.State

property State: **String**;

property AnsiString State;

Description

Returns the ODBC error state of the current ODBC error as a 5 character string. For a complete list of ODBC error states and their descriptions, see the ODBC documentation. The native database error codes are all mapped into these error states.

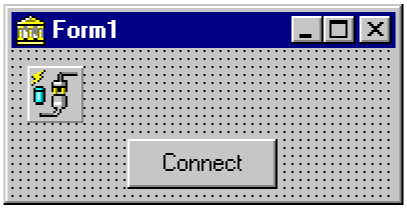
Example:

State	**Description**
IM002	Data source name not found and no default driver specified.

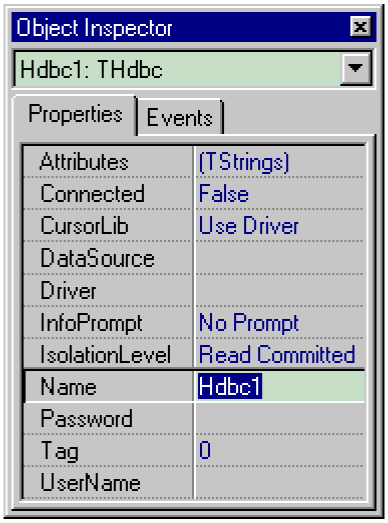
EODBC.Example

In this example we are going to trap and display an ODBC error using the Detail dialog of the exception class EODBC. To do this we throw an THdbc component on the form, leaving its properties at default values. In the OnClick event of a button, we are going to do a connect, which will fail because we haven't specified a DataSource to use in the THdbc component.

Form



Object Inspector



Hdbc1: THdbc

Attributes	(TStrings)
Connected	False
CursorLib	Use Driver
DataSource	
Driver	
InfoPrompt	No Prompt
IsolationLevel	Read Committed
Name	Hdbc1
Password	
Tag	0
UserName	

Delphi Code

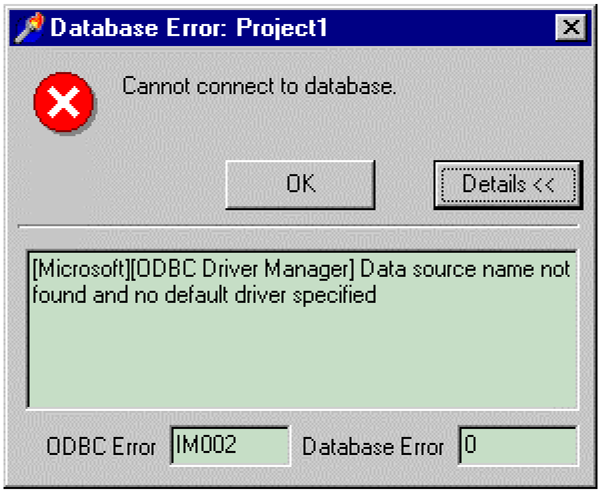
```
procedure TForm1.Button1Click(Sender: TObject);
begin
  try
    //try to make a connection
    Hdbc1.Connect;
  except
    on E: EODBC do
      //display the error if the connection fails
      E.Detail('Cannot connect to database.');
  end;
end;
```

C++ Code

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  try {
    //try to make a connection
    Hdbc1->Connect();
  }
  catch (EODBC *E) {
    //display the error if the connection fails
    E->Detail("Cannot connect to database.");
  }
}
```

Result

When you run the program and click on the Connect button, the ODBC Driver Manager will return the followiing error (click on the detail button to see the ODBC error message):



Cannot connect to database.

[Microsoft][ODBC Driver Manager] Data source name not found and no default driver specified

ODBC Error: IM002

Database Error: 0

TODBCError

Hierarchy Properties Methods Events Example

Unit OCL;

Description

The ODBC error class is used to validate the return codes of ODBC functions and raise ODBC errors if necessary. An instance of this class always exists as the GlobalHenv.Error, so it isn't necessary to create one as a stand-alone class.

TODBCError.OnConnectionFailure

property OnConnectionFailure: TNotifyEvent;

property TNotifyEvent OnConnectionFailure;

Description

This event is called when a connection failure occurs. It can be used to re-connect and re-open your queries, or apply manual connection resolution if needed. The Owner parameter of this event will be the THbase descendant in which the failure was detected. See the Resolve property for more details.

TODBCError.OnException

TODBCExceptionEvent = **function** (Sender: THbase;
                                                     E: EODBC): **Boolean of object**;

property OnException: TODBCExceptionEvent;

bool (**closure*** TODBCExceptionEvent)(THbase* Sender,
                                                           EODBC* E);

property TODBCExceptionEvent OnException;

Description

This event is called each time an ODBC error occurs. It is used to inspect an ODBC exception "E" returned by a handle class "Sender" before returning True to raise the exception or False to prevent the exception from being raised.

TODBCError.RaiseError

procedure RaiseError(AOwner: THbase;
                                RetCode: SQLRETURN);

void RaiseError(THbase* AOwner,
                        SQLRETURN RetCode);

Description

Used to raise the EODBC exception when ODBC functions are called externally from the ODBCExpress library. To guarantee that the correct ODBC error is returned, a call to this method must be made to raise an EODBC exception class, rather than raising it directly.

The AOwner parameter passed to this method must be the object whose handle was passed to the ODBC function and the RetCode parameter passed to this method is the return code of the ODBC function.

TODBCError.RaiseSoftErrors

property RaiseSoftErrors: **Boolean**;

property bool RaiseSoftErrors;

Description

Used to control whether soft ODBC errors must be raised as EODBC exceptions or not. The default value is False, since by default no soft ODBC errors need to be raised as exceptions. The value of this property influences the level of error testing done by the Success method. The SuccessOnly method on the other hand always treats soft ODBC errors as exceptions, irrespective of the value of the RaiseSoftErrors property.

TODBCError.Resolve

property Resolve: **Boolean**;

property bool Resolve;

Description

Used to recover from a broken connection (for example caused by a communication link failure) without re-starting your application. The default value is True.

When a connection failure occurs, the THenv.Resolve method or the THdbc.Resolve method will be called (depending on where the failure occurred) to place all the ODBCExpress components in the correct, closed state. The OnConnectionFailure event will then be called, which you can then use to re-connect and re-open your queries if you wish. If you set this property to False, the OnConnectionFailure event will still be called, however no connection resolution will be done beforehand. You can then manually call either the THenv.Resolve or THdbc.Resolve methods if you wish to recover from the connection failure.

When the ODBCExpress units are unloaded from memory, connection resolution is automatically applied to avoid any possible exceptions which could interfere with this process.

TODBCError.Success

function Success(RetCode: SQLRETURN): **Boolean**;

bool Success(SQLRETURN RetCode);

Description

Used to check if the return code of an ODBC function signifies the successful execution of the function. This is done by returning the result of the following boolean expression:

Delphi Code

```
if RaiseSoftErrors then
  Result:= RetCode = SQL_SUCCESS
else
  Result:= (RetCode = SQL_SUCCESS) or (RetCode = SQL_SUCCESS_WITH_INFO);
```

C++ Code

```
if (RaiseSoftErrors) {
  Return (RetCode == SQL_SUCCESS);
}
else {
  Return ((RetCode == SQL_SUCCESS) | (RetCode == SQL_SUCCESS_WITH_INFO));
}
```

TODBCError.SuccessOnly

function SuccessOnly(RetCode: SQLRETURN): **Boolean**;

bool SuccessOnly(SQLRETURN RetCode);

Description

Used to check if the return code of an ODBC function signifies the successful execution of the function. This is done by returning the result of the following boolean expression:

Delphi Code

```
Result:= RetCode = SQL_SUCCESS;
```

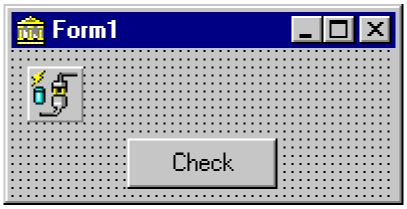
C++ Code

```
Return (RetCode == SQL_SUCCESS);
```

This is a stronger check than the check done by the Success method. Normally the Success method is used to validate the return code of an ODBC function, however the SuccessOnly method is used in special cases where stronger validation is required.

TODBCError Example

In this example, we attemp to determine if the SQLSetPos ODBC function (used to do positional operations such as inserts, updates and deletes) is supported by making a call to the SQLGetFunctions with the SQL_API_SQLSETPOS constant passed as a parameter. After the function call, we test the return code of the function, and if the function didn't execute successfully, we raise on ODBC exception. To do this, we make use of the global THenv component GlobalHenv.



Delphi Code

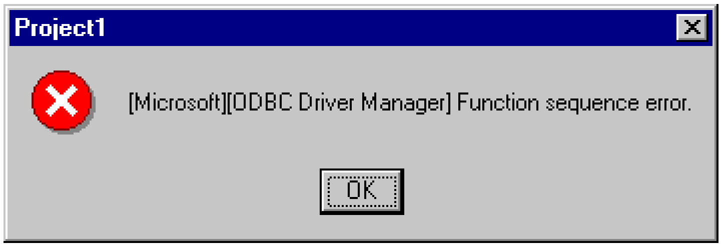
```
procedure TForm1.Button1Click(Sender: TObject);
var
  RetCode: SQLRETURN;
  Supported: SQLUSMALLINT;
begin
  //call the ODBC function
  RetCode:= SQLGetFunctions(Hdbc1.Handle, SQL_API_SQLSETPOS, @Supported);
  if not GlobalHenv.Error.Success(RetCode) then
    //raise an ODBC error if the function fails
    GlobalHenv.Error.RaiseError(Hdbc1, RetCode);

  if Supported = 0 then
    ShowMessage('Supported')
  else
    ShowMessage('Not Supported');
end;
```

C++ Code

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  SQLRETURN RetCode;
  SQLUSMALLINT Supported;

  //call the ODBC function
  RetCode = SQLGetFunctions(Hdbc1->Handle(), SQL_API_SQLSETPOS, &Supported);
  if (!GlobalHenv->Error->Success(RetCode)) {
    //raise an ODBC error if the function fails
    GlobalHenv->Error->RaiseError(Hdbc1, RetCode);
  }

  if (Supported == 0) {
    ShowMessage("Supported");
  }
  else {
    ShowMessage("Not Supported");
  }
}
```

Result

Because the function SQLGetFunctions only work when the THdbc component is connected to a DataSource, we will get a "Function Sequence Error" raised by the RaiseError method when we click on the Check button, since we havn't connected to the DataSource:



[Microsoft][ODBC Driver Manager] Function sequence error.

THbase

Hierarchy Methods Events

Unit OCL;

Description

This component is the ancestor of the three ODBCExpress Handle Components, THenv, THdbc and THstmt. It contains the common functionality of the three components, and is mainly used to test if an object is of one of the three handle components.

THbase.Init

function Init: **Boolean**; **virtual**;

virtual bool Init(**void**);

Description

Initializes the ODBC handles contained in the current handle component and in all the parent handle components of the current handle component. This method is automatically called when needed.

THbase.OnInit

property OnInit: TNotifyEvent;

property TNotifyEvent OnInit;

Description

This event is called on each call to the Init method of one of the three handle components.

THbase.OnTerminate

property OnTerminate: TNotifyEvent;

property TNotifyEvent OnTerminate;

Description

This event is called on each call to the Terminate method of one of the three handle components.

THbase.Terminate

function Terminate: **Boolean**; **virtual**;

virtual bool Terminate(**void**);

Description

Terminates all the ODBC handles contained in the current handle component and in all the children handle components of the current handle component. This method is automatically called when the handle component is destroyed.

Note:

This method rarely needs to be called by the developer. The handle contained in a handle component is automatically reset when the component is re-used. Only in rare circumstances where the resetting of a handle is not enough to avoid a function sequence error is this method used, and then mostly with the THstmt component.

THenv

Hierarchy   Properties   Methods

Unit OCL;

Description

The environment component is used to establish and maintain an environment handle for the application. The component is at the root of the handle tree and only one is needed per application. For this reason a default environment component, called GlobalHenv, is created in this unit for each application, which means that generally there's no need to create an additional environment component for use in your application.

A global ThreadedHenv boolean variable will, when set to True, allow multiple GlobalHenv components to be automatically distributed across threads. This is to allow extended functionality such as providing a separate connection pool for each new thread. Be default a single GlobalHenv component is used across multiple threads, since ODBC is thread-safe at environment level for all ODBC drivers. If however multiple GlobalHenv variables are used, then the global ClearHenv method must be called on exit of each non-main thread to free the GlobalHenv for that specific thread.

GlobalHenv Declarations

Delphi Code

```
var
  ThreadedHenv: Boolean = False;

function GlobalHenv: THenv;
procedure ClearHenv;
```

C++ Code

```
bool ThreadedHenv = false;

THenv* GlobalHenv(void);
void ClearHenv(void);
```

THenv.Active

property Active: **Boolean**;

property bool Active;

Description

Returns True if the environment handle is active, otherwise returns False. This property is rarely queried, since an exception is raised in the component if the environment handle is not successfully created.

THenv.ConnectionPooling

TConnectionPooling = (cpDefault, cpOff, cpOnePerDriver, cpOnePerEnv);

property ConnectionPooling: TConnectionPooling;

TConnectionPooling {cpDefault, cpOff, cpOnePerDriver, cpOnePerEnv};

property TConnectionPooling ConnectionPooling;

Description

Used to enable connection pooling. The default value is cpDefault. The property can have one of the following values:

cpDefault	Allows connection pooling to be controlled externally. This option is useful when using ODBCExpress in an environment which controls connection pooling externally (e.g. in a Microsoft Transaction Server environment). However when there is no external agent involved, connection pooling is always off.
cpOff	Sets connection pooling off.
cpOnePerDriver	Sets one connection pool per ODBC driver.
cpOnePerEnv	Sets one connection pool per ODBC environment.

Connection pooling speeds up your average connection time and reduces the number of physical database connections used. When enabled, the ODBC Driver Manager keeps a pool of connections, and when it then receives a request to establish a new connection, it first checks if an appropriate connection is not in the pool before a new connection is established. If the ODBC Driver Manager finds a suitable connection in the connection pool which matches a connect request, it will use this connection rather than creating a new connection, resulting in virtually instantaneous connection time. In this way a single connection can be shared by any number of users or applications. Connections are also kept in the connection pool for a long time after it's not in use any more, only dropping the connection after its 'not-in-use' time in the pool has expired or when the application with the ODBC environment controlling the connection pool is closed. In other words connection pooling is also advantages in a client application where a connection to a database is established and dropped multiple times. It will reduce re-connection times drastically.

THenv.Error

property Error: TODBCError;

property TODBCError* Error;

Description

An instance of the TODBCError class which is used to validate the return code of ODBC functions and raise ODBC exceptions. Generally the instance created as the Error property of the global environment component GlobalHenv is used in applications.

THenv.Handle

property Handle: SQLHENV;

property SQLHENV Handle;

Description

Returns the ODBC environment handle contained in the component. This handle can be used to make other native ODBC function calls which are not directly supported by the environment component.

THenv.Resolve

procedure Resolve;

void Resolve(**void**);

Description

This method is used to resolve a connection failure at environment level. If the TODBCError.Resolve property is False, this method is used (probably from within THenv.OnConnectionFailure event) to manually recover from a connection failure. If the TODBCError.Resolve property is True, this method will automatically be called if the connection failure occurs at environment level. See the TODBCError.Resolve property for more details.

THenv.Terminate

function Terminate: **Boolean**; **override**;

virtual bool Terminate(**void**);

Description

Terminates all the children connection components of the environment component by calling the Terminate method of each child connection component. This method is automatically called when the environment component is destroyed.

THenv.WinHandle

property WinHandle: SQLHWND;

property SQLHWND WinHandle;

Description

This read-only property returns the windows handle used for dialog boxes in ODBCExpress (e.g. the Detail Error Dialog and the ODBC Login Dialog). If the WinParent property is set to a windows control, ODBCExpress will use the handle of this windows control for the dialog boxes, otherwise ODBCExpress will use the handle of Application or the handle Application.MainForm (in this order). WinHandle will return a null handle if all the previous handles are null.

THenv.WinParent

property WinParent: TWinControl;

property TWinControl* WinParent;

Description

This property can be set if necessary in non-visual environments (such as DLL's) where WinHandle otherwise returns a null handle, to allow ODBCExpress to display the dialog boxes. If no dialog boxes are needed in the DLL, or Application.MainForm has a value, then it's not necessary to set this property.

THdbc

Hierarchy Properties Methods Events

Unit OCL;

Description

The connection component is used to establish and maintain a connection with a DataSource. Different connections in the same application have no influence on each other and acts as separate data access applications. It is therefore possible to spin of different connections on seperate threads. For example, a timely query at a DataSource can be spun off on one thread, while the user can do other queries at the same (or different) DataSource using another connection component.

THdbc.Active

property Active: **Boolean**;

property bool Active;

Description

Returns True if the connection handle is active, otherwise returns False. However, this property is rarely queried, since an exception is raised in the component if the connection handle is not successfully created.

THdbc.AddDriver

function AddDriver(ADriver: **String**): TDriverPtr;

TDriverPtr AddDriver(**AnsiString** ADriver);

Description

Adds a new driver to the list of driver profiles.

Driver profiles allow usage of driver specific settings. Each entry in the driver profile list consists of the following record:

Delphi Code

```
type
  TDriverPtr = ^TDriverRec;
  TDriverRec = record
    //description of the driver as returned by the CurrentDriver method
    Desc: String;
    //the precision (parameter size) values for the following data types
    PS_SQL_CHAR,
    PS_SQL_VARCHAR,
    PS_SQL_LONGVARCHAR,
    PS_SQL_BINARY,
    PS_SQL_VARBINARY,
    PS_SQL_LONGVARBINARY,
    PS_SQL_DECIMAL,
    PS_SQL_NUMERIC,
    PS_SQL_TYPE_TIMESTAMP: SQLUINTEGER;

    //the scale (decimal digits) values for the following data types
    DD_SQL_DECIMAL,
    DD_SQL_NUMERIC,
    DD_SQL_TYPE_TIMESTAMP: SQLSMALLINT;
  end;
```

C++ Code

```
typedef TDriverRec* TDriverPtr;
struct TDriverRec {
  //description of the driver as returned by the CurrentDriver method
  AnsiString Desc;

  //the precision (parameter size) values for the following data types
  SQLUINTEGER PS_SQL_CHAR;
  SQLUINTEGER PS_SQL_VARCHAR;
  SQLUINTEGER PS_SQL_LONGVARCHAR;
  SQLUINTEGER PS_SQL_BINARY;
  SQLUINTEGER PS_SQL_VARBINARY;
  SQLUINTEGER PS_SQL_LONGVARBINARY;
  SQLUINTEGER PS_SQL_DECIMAL;
  SQLUINTEGER PS_SQL_NUMERIC;
  SQLUINTEGER PS_SQL_TYPE_TIMESTAMP;

  //the scale (decimal digits) values for the following data types
  SQLSMALLINT DD_SQL_DECIMAL;
  SQLSMALLINT DD_SQL_NUMERIC;
  SQLSMALLINT DD_SQL_TYPE_TIMESTAMP;
};
```

A default list of driver records with settings are automatically added. The "Desc" fields of this driver list are as follows:

```
  //the default settings used by all drivers
  Default

  //some modified settings for drivers shipped with MDAC
  Microsoft SQL Server;sqlsrv32.dll
  Oracle;msorcl32.dll
  Access;odbcjt32.dll
  dBase;odbcjt32.dll
  Paradox;odbcjt32.dll
  Visual FoxPro;vfpodbc.dll
  FoxPro;odbcjt32.dll
  //some modified settings for drivers shipped with the following DBMS's
  Oracle;msorcl32.dll
  SQL Server;sysybnt.dll
  Adaptive Server Anywhere;dbodbc6w.dll
  Informix;iclit09a.dll
  InterBase;iscdrv32.dll
```

A number of methods allow you to manipulate this driver list if necessary.

```
  procedure RefreshDrivers;
  procedure ClearDrivers;
  function AddDriver(ADriver: String): TDriverPtr;
  procedure RemoveDriver(ADriver: String);
  function GetDriver(ADriver: String): TDriverPtr;
  function CurrentDriver: String;
  function IsDriver(ADrivers: array of String): Boolean;
```

The record returned by AddDriver contains default values as contained by the Default driver record, so you only need to change the record fields which are different from the default, e.g.:

Delphi Code

```
with Hdbc1.AddDriver(CurrentDriver)^ do
begin
  PS_SQL_CHAR:= 2000;

  //19+Fractional
  PS_SQL_TYPE_TIMESTAMP:= 25;

  //Fractional
  DD_SQL_TYPE_TIMESTAMP:= 5;
end;
```

C++ Code

```
TDriverPtr Driver;
Driver = Hdbc1->AddDriver(Hdbc1->CurrentDriver());

Driver->PS_SQL_CHAR = 2000;

//19+Fractional
Driver->PS_SQL_TYPE_TIMESTAMP = 25;

//Fractional
Driver->DD_SQL_TYPE_TIMESTAMP = 5;
```

The global, non-connection specific Fractional variable, which where used to calculate the above timestamp precision and scale, are replaced by the above connection specific option.

Example:

Delphi Code

```
//adds a driver record for the current driver connected to
```

```
Hdbc1.AddDriver(Hdbc1.CurrentDriver);
```

C++ Code

```
//adds a driver record for the current driver connected to
Hdbc1->AddDriver(Hdbc1->CurrentDriver());
```

THdbc.AfterConnect

property AfterConnect: TNotifyEvent;

property TNotifyEvent AfterConnect;

Description

This event is called after a connection with a database is established.

THdbc.AfterDisconnect

property AfterDisconnect: TNotifyEvent;

property TNotifyEvent AfterDisconnect;

Description

This event is called after a connection with a database is dropped.

THdbc.Attributes

property Attributes: TStrings;

property TStrings* Attributes;

Description

Used to specify additional attributes used during connection to a database. Apart from the standard attributes DataSource, UserName, Password and Driver which can be specified as properties of the connection component, this property is used to specify DataSource-specific attributes not directly supported by the connection component. With some ODBC Drivers it's also possible to provide enough attributes to connect to a database without specifying a DataSource, as the following example demonstrates:

Example:

Delphi Code

```
with Hdbc1 do
begin
  //set general attributes, without setting the DataSource attribute
  Driver:= 'SQL Server';
  UserName:= 'SA';
  Password:= 'PW';
  //add the server and database attributes needed to connect to the database
  Attributes.Clear;
  Attributes.Add('SERVER=JUBILEE');
  Attributes.Add('DATABASE=STUDENTDB');

  //establish the DataSource-less connection
  Connect;
end;
```

C++ Code

```
  //set general attributes, without setting the DataSource attribute
  Hdbc1->Driver = "SQL Server";
  Hdbc1->UserName = "SA";
  Hdbc1->Password = "PW";

  //add the server and database attributes needed to connect to the database
  Hdbc1->Attributes->Clear();
  Hdbc1->Attributes->Add("SERVER=JUBILEE");
  Hdbc1->Attributes->Add("DATABASE=STUDENTDB");

  //establish the DataSource-less connection
  Hdbc1->Connect();
```

THdbc.BeforeConnect

property BeforeConnect: TNotifyEvent;

property TNotifyEvent BeforeConnect;

Description

This event is called before a connection with a database is established.

THdbc.BeforeDisconnect

property BeforeDisconnect: TNotifyEvent;

property TNotifyEvent BeforeDisconnect;

Description

This event is called before a connection with a database is dropped.

THdbc.ClearDrivers

procedure ClearDrivers;

void ClearDrivers(**void**);

Description

Clears the driver profile list. See the AddDriver property for more details on driver profiles.

THdbc.Commit

procedure Commit;

void Commit(**void**);

Description

Commits the current transaction at the DataSource and implicitly starts a new transaction.

THdbc.Connect

procedure Connect;

void Connect(**void**);

Description

Connects the component to the specified DataSource, making use of the DataSource, UserName, Password, Driver and other Attributes to establish the connection. This method will automatically be called when a child statement component is used without manually calling this method first.

THdbc.Connected

property Connected: **Boolean**;

property bool Connected;

Description

The Connected property can be used to read or set the connected state of the component:

Delphi Code

"Connected:= True" is equivalent to "Hdbc.Connect".
"Connected:= False" is equivalent to "Hdbc.Disconnect".

C++ Code

"Connected = True" is equivalent to "Hdbc->Connect()".
"Connected = False" is equivalent to "Hdbc->Disconnect()".

THdbc.ConnectionPooling

property ConnectionPooling: **Boolean**;

property bool ConnectionPooling;

Description

Used to set whether connection pooling must be used or not. The default value is False.

Connection pooling can be enabled either by setting the ConnectionPooling property of theTHdbc component or the ConnectionPooling property of the GlobalHenv component toTrue. Setting THenv.ConnectionPooling or THdbc.ConnectionPooling to True will affect all connection components used in the application. Connection pooling is actually an environment-level function, however it is also available at THdbc level to allow setting of this property via the Object Inspector.

THdbc.Core

property Core: **Boolean**;

property bool Core;

Description

This property can be used to determine whether the ODBC Driver used can make use of normal parameter binding (using the Hstmt.BindParam method in which the database datatypes of parameters are detected) or only core parameter binding (using the Hstmt.BindParamCore method in which database datatypes of parameters are supplied by ODBCExpress). This property can also be set to force an ODBC Driver which can make use of normal parameter binding to now make use of core parameter binding instead.

Note:

Querying of this property is normally done internally by ODBCExpress, and setting of the property is also done by ODBCExpress on connection to a database, so it shouldn't be necessary to ever read or set this property manually. Only in a case where normal parameter binding is used and ODBC doesn't return the correct database datatype for the parameter being bound is it necessary to force core parameter binding by setting the Core property True. Instead of setting the Core property to True, core parameter binding can also be done by calling the BindParamCore method directly instead of one of the other Bind methods (BindString, BindInteger, etc.) of the THstmt component.

THdbc.CurrentDriver

function CurrentDriver: **String**;

AnsiString CurrentDriver(**void**);

Description

Returns the description of the current driver connected to, for use with the driver profile methods. See the AddDriver property for more details on driver profiles.

THdbc.CursorLib

property CursorLib: SQLUINTEGER;

property SQLUINTEGER CursorLib;

Description

If your ODBC Driver doesn't support scrollable cursors, you can make use of the ODBC Cursor Library which provides you with a static scrollable cursor by making use of front-end row buffering. This property can be set to one of the following values:

SQL_CUR_USE_DRIVER	Use the ODBC Driver's scrolling capabilities (default).
SQL_CUR_USE_ODBC	Use the ODBC Cursor Library to provide scrolling capabilities.
SQL_CUR_USE_IF_NEEDED	Use the ODBC Cursor Library only if the ODBC driver doesn't support Scrollable cursors.

The cursor library not only provides you with a scrollable cursor, but also enables the ODBC Driver to perform positional operations (updates, deletes, etc.) and fetch multiple rows at a time.

Note:

1. The ODBC Cursor Library only supports static cursors, and concurrency is achieved comparing values. This means that the following two properties of the Hstmt class must be set as follows for the cursor library to work:

CursorType	SQL_CURSOR_STATIC
ConcurrencyType	SQL_CONCUR_VALUES

2. When doing a positional update while using the Cursor Library, make sure that the columns that are being updated uniquely identifies the row that is being updated.

Example:

Delphi Code

```
//enable the cursor library for an ODBC Driver which doesn't support scrollable cursors directly
Hdbc1.CursorLib:= SQL_CUR_USE_ODBC;

//set the cursor type and concurrency type to the values which work with the cursor library
Hstmt1.CursorType:= SQL_CURSOR_STATIC;
Hstmt1.ConcurrencyType:= SQL_CONCUR_VALUES;

with Hstmt1 do
begin
  //build the select statement to also include the full primary key, which is SNo in this case
  SQL:= 'SELECT SNo, SName, SGender FROM Students';
  Prepare;
  Execute;
  FetchLast;
end;
```

C++ Code

```
//enable the cursor library for an ODBC Driver which doesn't support scrollable cursors directly
Hdbc1->CursorLib = SQL_CUR_USE_ODBC;

//set the cursor type and concurrency type to the values which work with the cursor library
```

```
Hstmt1->CursorType = SQL_CURSOR_STATIC;
Hstmt1->ConcurrencyType = SQL_CONCUR_VALUES;

//build the select statement to also include the full primary key, which is SNo in this case
Hstmt1->SQL = "SELECT SNo, SName, SGender FROM Students";
Hstmt1->Prepare();
Hstmt1->Execute();
Hstmt1->FetchLast();
```

THdbc.DataSource

property DataSource: **String**;

property AnsiString DataSource;

Description

Used to specify the ODBC DataSource that must be connected to. The default value is an empty string. ODBC DataSources can be created and maintained by the ODBC Administrator, usually located in the Control Panel.

Example:

Delphi Code

```
Hdbc1.DataSource:= 'dsnStudentData';
```

C++ Code

```
Hdbc1->DataSource = "dsnStudentData";
```

THdbc.Disconnect

procedure Disconnect;

void Disconnect(**void**);

Description

Disconnects the component from the DataSource. This method is automatically called when the component is destroyed.

THdbc.DoAfterConnect

procedure DoAfterConnect; **virtual**;

virtual void DoAfterConnect(**void**);

Description

Calls the OnAfterConnect event.

THdbc.DoAfterDisconnect

procedure DoAfterDisconnect; **virtual**;

virtual void DoAfterDisconnect(**void**);

Description

Calls the OnAfterDisconnect event.

THdbc.DoBeforeConnect

procedure DoBeforeConnect; **virtual**;

virtual void DoBeforeConnect(**void**);

Description

Calls the OnBeforeConnect event.

THdbc.DoBeforeDisconnect

procedure DoBeforeDisconnect; **virtual**;

virtual void DoBeforeDisconnect(**void**);

Description

Calls the OnBeforeDisconnect event.

THdbc.Driver

property Driver: **String**;

property AnsiString Driver;

Description

Optional property used during the connection to the DataSource. It indicates which ODBC Driver to use during the connection, and is the ODBC Driver Name string contained in the registry, which is displayed by the ODBC Administrator in the Control Panel. A list of ODBC Driver Names registered on a machine can be returned as the Drivers property of the TOEAdministrator component.

Example:

Delphi Code

```
//indicate the Microsoft SQL Server ODBC Driver
Hdbc1.Driver:= 'SQL Server';
```

C++ Code

```
//indicate the Microsoft SQL Server ODBC Driver
Hdbc1->Driver = "SQL Server";
```

THdbc.EndTransact

procedure EndTransact;

void EndTransact(**void**);

Description

Commits the current transaction at the DataSource and places the connection component back into auto-commit mode.

THdbc.GetDriver

function GetDriver(ADriver: **String**): TDriverPtr;

TDriverPtr GetDriver(**AnsiString** ADriver);

Description

Returns the record for the driver ADriver in the driver profile list, or nil if not found. See the AddDriver property for more details on driver profiles.

Example:

Delphi Code

```
var
  Driver: TDriverPtr;
begin
  //return the driver record of the current driver connected to
  Driver:= Hdbc1.GetDriver(Hdbc1.CurrentDriver);
end;
```

C++ Code

```
TDriverPtr Driver;

//return the driver record of the current driver connected to
Driver = Hdbc1->GetDriver(Hdbc1->CurrentDriver());
```

THdbc.GetFunction

function GetFunction(FunctionType: SQLUSMALLINT): **Boolean**;

bool GetFunction(SQLUSMALLINT FunctionType);

Description

Used to determine if a specific ODBC function is supported by the current ODBC Driver. See the ODBC documentation for details on the SQLGetFunctions ODBC function.

THdbc.GetInfoInteger

function GetInfoInteger(InfoType: SQLUSMALLINT): SQLINTEGER;

SQLUINTEGER GetInfoInteger(SQLUSMALLINT InfoType);

Description

Used to retrieve information about the current ODBC Driver as an integer. See the ODBC documentation for details on the SQLGetInfo ODBC function.

THdbc.GetInfoSmallint

function GetInfoSmallint(InfoType: SQLUSMALLINT): SQLUSMALLINT;

SQLUSMALLINT GetInfoSmallint(SQLUSMALLINT InfoType);

Description

Used to retrieve information about the current ODBC Driver as a smallint. See the ODBC documentation for details on the SQLGetInfo ODBC function.

THdbc.GetInfoString

function GetInfoString(InfoType: SQLUSMALLINT): **String**;

AnsiString GetInfoString(SQLUSMALLINT InfoType);

Description

Used to retrieve information about the current ODBC Driver as a string. See the ODBC documentation for details on the SQLGetInfo ODBC function.

THdbc.Handle

property Handle: SQLHDBC;

property SQLHDBC Handle;

Description

Returns the ODBC connection handle contained in the component. This handle can be used to call other ODBC functions which are not directly supported by the connection component.

THdbc.InfoPrompt

property InfoPrompt: SQLUSMALLINT;

property SQLUSMALLINT InfoPrompt;

Description

Used to set the level of user prompting that must be done to establish a connection. The default value is SQL_DRIVER_NOPROMPT. It can have one of the following values:

SQL_DRIVER_NOPROMPT	An attemp will be made to connect to the database with the given data source, user name and password properties. If it fails, an ODBCError exception will be raised.
SQL_DRIVER_PROMPT	A dialog box will be displayed by ODBC, using the values of the three properties data source, user name and password (if any) as initial values. When the user exits the dialog box, the driver connects to the data source.
SQL_DRIVER_COMPLETE	If the three properties data source, user name and password contains enough information to connect to the database, then the connection will be made without prompting the user, otherwise the user will be prompted to fill in the missing information.
SQL_DRIVER_COMPLETE_REQUIRED	The same as SQL_DRIVER_COMPLETE, except that the controls for any information which is not required to connect to the data source will be disabled if the user is prompted.

THdbc.InTransaction

property InTransaction: **Boolean**;

property bool InTransaction;

Description

Allows you to determine whether you are in manual commit mode or not. This property always returns True between calls to StartTransact and EndTransact.

THdbc.IsDriver

function IsDriver(ADrivers: array of **String**): **Boolean**;

bool IsDriver(**const AnsiString*** ADrivers, **const int** ADrivers_Size);

Description

Checks if the current driver connected to matches any driver description in the array ADrivers, for use with the driver profile methods. See the AddDriver property for more details on driver profiles.

THdbc.IsolationLevel

property IsolationLevel: SQLUINTEGER;

property SQLUINTEGER IsolationLevel;

Description

Used to set the isolation between transactions at the DataSource for the current connection. The default value is SQL_TXN_READ_COMMITTED. It can have one of the following values:

SQL_TXN_READ_UNCOMMITTED	Dirty reads, non-repeatable reads and phantom reads are possible.
SQL_TXN_READ_COMMITTED	Dirty reads are not possible. Nonrepeatable reads and phantom reads are possible.
SQL_TXN_REPEATABLE_READ	Dirty reads and nonrepeatable reads are not possible. Phantom reads are possible.
SQL_TXN_SERIALIZABLE	Transactions are serializable. Dirty reads, nonrepeatable reads and phantom reads are not possible.

The following terms are used to define the isolation levels:

Dirty Read	Transaction 1 changes a row. Transaction 2 reads the changed row before transaction 1 commits the change. If transaction 1 rolls back the change, transaction 2 will have read a row that is considered to have never existed.
Nonrepeatable Read	Transaction 1 reads a row. Transaction 2 updates or deletes that row and commits this change. If transaction 1 attempts to reread the row, it will receive different row values or discover that the row has been deleted.
Phantom Read	Transaction 1 reads a set of rows that satisfy some search criteria. Transaction 2 inserts a row that matches the search criteria. If transaction 1 reexecutes the statement that read the rows, it receives a different set of rows.

THdbc.LoginTimeOut

property LoginTimeOut: SQLUINTEGER;

property SQLUINTEGER LoginTimeOut;

Description

Used to set the number of seconds after which an attemp to connect to a datasource must timeout. A value of 0 indicates that no timeout must occur. The default value is ODBC Driver dependent.

Example:

Delphi Code

```
//set timeout to occur after 20 seconds
Hdbc1.LoginTimeOut:= 20;
Hdbc1.Connect;
```

C++ Code

```
//set timeout to occur after 20 seconds
Hdbc1->LoginTimeOut = 20;
Hdbc1->Connect();
```

THdbc.Password

property Password: **String**;

property AnsiString Password;

Description

Used to specify the access code of the account that must be connected to. The default value is an empty string.

THdbc.RefreshDrivers

procedure RefreshDrivers;

void RefreshDrivers(**void**);

Description

Clears the driver profile list and adds the default list of drivers. See the AddDriver property for more details on driver profiles.

THdbc.RemoveDriver

procedure RemoveDriver(ADriver: **String**);

void RemoveDriver(**AnsiString** ADriver);

Description

Removes a driver from the driver profile list. See the AddDriver property for more details on driver profiles.

Example:

Delphi Code

```
//remove the driver record of the current driver connected to
Hdbc1.RemoveDriver(Hdbc1.CurrentDriver);
```

C++ Code

```
//remove the driver record of the current driver connected to
Hdbc1->RemoveDriver(Hdbc1->CurrentDriver());
```

THdbc.Resolve

procedure Resolve;

void Resolve(**void**);

Description

This method is used to resolve a connection failure at connection level. If the TODBCError.Resolve property is False, this method is used (probably from within THenv.OnConnectionFailure event) to manually recover from a connection failure. If the TODBCError.Resolve property is True, this method will automatically be called if the connection failure occurs at connection level. See the TODBCError.Resolve property for more details.

THdbc.Rollback

procedure Rollback;

void Rollback(**void**);

Description

Rolls back the current transaction at the DataSource and implicitly starts a new transaction.

THdbc.StartTransact

procedure StartTransact;

void StartTransact(**void**);

Description

The connection component is in auto-commit mode by default. In other words, all operations done at the DataSource are automatically committed. This method takes the component out of auto-commit mode and starts a new transaction.

A new transaction is started after each call to StartTransact, Commit and RollBack, while a transaction is ended by each call to Commit, RollBack and EndTransact. Note that a transaction only starts at the first statement call made after StartTransact, Commit or RollBack was called. Also, calling Commit, RollBack or EndTransact without preceding it with transactional operations has no effect and won't cause any errors. Always match each StartTransact with an EndTransact (preferably as part of a try-finally statement).

Example:

Delphi Code

```
//start transaction
Hdbc1.StartTransact;

try

  { data access code }

  //commit transaction and start next transaction
  Hdbc1.Commit;
  { data access code which fails }

  //roll back transaction and start next transaction
  Hdbc1.Rollback;

  { data access code }

finally

  //end transaction
  Hdbc1.EndTransact;

end;
```

C++ Code

```
//start transaction
Hdbc1->StartTransact();

try {

  //data access code

  //commit transaction and start next transaction
  Hdbc1->Commit();

  //data access code which fails

  //roll back transaction and start next transaction
  Hdbc1->Rollback();

  //data access code

}
catch ( ... ) {
}

//end transaction
Hdbc1->EndTransact();
```

THdbc.Terminate

function Terminate: **Boolean**; **override**;

virtual bool Terminate(**void**);

Description

Terminates all the children statement components of the connection component by calling the Terminate method of each child statement component. This method is automatically called when the connection component is destroyed. This method also disconnects the connection component from the DataSource and terminates its connection with ODBC.

THdbc.Tracing

property Tracing: **Boolean**;

property bool Tracing;

Description

If this property is set to True, all ODBC calls will be traced and written out to the file TRACE.OE in the same directory as the application. This is useful to pinpoint problem areas in ODBC Drivers.

Note:

Tracing should only be used during debugging if necessary, since it slows down the performance of your application and might violate your database security if included in your final application.

THdbc.UserName

property UserName: **String**;

property AnsiString UserName;

Description

Used to specify the account at the DataSource that must be connected to. The default value is an empty string.

THdbc.Version

property Version: **String**;

property AnsiString Version;

Description

This read-only property returns the current version number of the ODBCExpress components. If the value at design-time does not reflect the value at run-time, you have a consistency problem between you design-time and run-time ODBCExpress code.

THstmt

Hierarchy Properties Methods Events

Unit OCL;

Description

The statement component is used to establish and maintain a statement handle and associate it with a connection component. This is the main functional component, which provides all the functionality for interaction with a DataSource. It automatically creates and maintains storage for all the columns in a result set. The component can also be transformed into a high speed data retrieval component, fetching large numbers of rows from the DataSource at a time.

This component makes use of core, level 1 and level 2 ODBC functionality to manipulate data at a DataSource. However, it is possible to bypass all level 2 functionality, which makes it possible for this component to work with ODBC Drivers with only a level 1 API conformance. Note that an ODBC Driver being a level 1 driver does not mean it does not support some or nearly all level 2 functionality.

THstmt.AbortQuery

procedure AbortQuery;

void AbortQuery(**void**);

Description

Used to cancel the processing of a query executed asynchronously. See the ExecAsync property for an example.

THstmt.Active

property Active: **Boolean**;

property bool Active;

Description

Returns True if the statement handle is active, otherwise returns False. However, this property is rarely queried, since an exception is raised in the component if the statement handle is not successfully created.

THstmt.AfterExecute

property AfterExecute: TNotifyEvent;

property TNotifyEvent AfterExecute;

Description

This event is called after each Execute made with the statement component.

THstmt.AfterFetch

property AfterFetch: TNotifyEvent;

property TNotifyEvent AfterFetch;

Description

This event is called after each Fetch made with the statement component.

THstmt.AfterPrepare

property AfterPrepare: TNotifyEvent;

property TNotifyEvent AfterPrepare;

Description

This event is called after each Prepare made with the statement component.

THstmt.BeforeExecute

property BeforeExecute: TNotifyEvent;

property TNotifyEvent BeforeExecute;

Description

This event is called before each Execute made with the statement component.

THstmt.BeforeFetch

property BeforeFetch: TNotifyEvent;

property TNotifyEvent BeforeFetch;

Description

This event is called before each Fetch made with the statement component.

THstmt.BeforePrepare

property BeforePrepare: TNotifyEvent;

property TNotifyEvent BeforePrepare;

Description

This event is called before each Prepare made with the statement component.

THstmt.BindBinary

procedure BindBinary(Param: SQLUSMALLINT;
                    var ParamValue: TMemoryStream);

void BindBinary(SQLUSMALLINT Param,
               TMemoryStream* &ParamValue);

Description

Used to bind a binary blob parameter in a SQL statement.

BindBinary(Param, ParamValue) is equivalent to
BindMemory(Param, ParamValue, True)

THstmt.BindBinaryByName

procedure BindBinaryByName(ParamName: **String**;
var ParamValue: TMemoryStream);

void BindBinaryByName(**AnsiString** ParamName,
TMemoryStream* &ParamValue);

Description

Performs the same function as the BindBinary method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindBinaryByName(ParamName, ...) is equivalent to BindBinary(ParamByName(ParamName), ...).

THstmt.BindBookmarks

property BindBookmarks: **Boolean**;

property bool BindBookmarks;

Description

To get some extra speed when you make use of result set cursors and bookmarks, bookmark columns will be bound as part of the result set when this property is True, instead of retrieving the bookmarks from the DataSource as needed. Of course if you're not making use of bookmarks (or not that much), then leaving BindBookmarks as False will give you better performance. The default value is False.

THstmt.BindByName

property BindByName: **Boolean**;

property bool BindByName;

Description

Used to bind stored procedure parameters by name rather than by position in the SQL statement. The default value is False. When set to True, you can declare the stored procedure parameters in any order in the SQL statement, using named parameters which are exactly the same as declared in the stored procedure.

THstmt.BindByte

```
procedure BindByte(Param: SQLUSMALLINT;
                              var ParamValue: Byte);

void BindByte(SQLUSMALLINT Param,
                    Byte &ParamValue);
```

Description

Used to bind a byte parameter in a SQL statement.

BindByte(Param, ParamValue) is similar to both
BindParam(Param, SQL_C_UTINYINT, @ParamValue) and
BindParamCore(Param, SQL_C_UTINYINT, @ParamValue, SQL_TINYINT)

THstmt.BindByteByName

```
procedure BindByteByName(ParamName: String;
                                       var ParamValue: Byte);

void BindButeByName(AnsiString ParamName,
                              Byte &ParamValue);
```

Description

Performs the same function as the BindByte method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindByteByName(ParamName, ...) is equivalent to BindByte(ParamByName(ParamName), ...).

THstmt.BindBytes

```
procedure BindBytes(Param: SQLUSMALLINT;
                               var ParamValue: array of Byte);

void BindBytes(SQLUSMALLINT Param,
                     Byte* ParamValue,
                     const int ParamValue_Size);
```

Description

Used to bind a byte parameter array in a SQL statement.

BindBytes(Param, ParamValue) is equivalent to
BindParams(Param, SQL_C_UTINYINT, @ParamValue, High(ParamValue)+1)

THstmt.BindBytesByName

```
procedure BindBytesByName(ParamName: String;
                                        var ParamValue: array of Byte);
```

```
void BindBytesByName(AnsiString ParamName,
                     Byte* ParamValue,
                     const int ParamValue_Size);
```

Description

Performs the same function as the BindBytes method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindBytesByName(ParamName, ...) is equivalent to BindBytes(ParamByName(ParamName), ...).

THstmt.BindCardinal

```
procedure BindCardinal(Param: SQLUSMALLINT;
                       var ParamValue: Cardinal);
```

```
void BindCardinal(SQLUSMALLINT Param,
                  Cardinal &ParamValue);
```

Description

Used to bind a cardinal parameter in a SQL statement.

BindCardinal(Param, ParamValue) is similar to both
BindParam(Param, SQL_C_ULONG, @ParamValue) and
BindParamCore(Param, SQL_C_ULONG, @ParamValue, SQL_INTEGER)

THstmt.BindCardinalByName

```
procedure BindCardinalByName(ParamName: String;
                             var ParamValue: Cardinal);
```

```
void BindCardinalByName(AnsiString ParamName,
                        Cardinal &ParamValue);
```

Description

Performs the same function as the BindCardinal method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindCardinalByName(ParamName, ...) is equivalent to
BindCardinal(ParamByName(ParamName), ...).

THstmt.BindCardinals

```
procedure BindCardinals(Param: SQLUSMALLINT;
                        var ParamValue: array of Cardinal);
```

```
void BindCardinals(SQLUSMALLINT Param,
                   Cardinal* ParamValue,
                   const int ParamValue_Size);
```

Description

Used to bind a cardinal parameter array in a SQL statement.

BindCardinals(Param, ParamValue) is equivalent to
BindParams(Param, SQL_C_ULONG, @ParamValue, High(ParamValue)+1)

THstmt.BindCardinalsByName

```
procedure BindCardinalsByName(ParamName: String;
                              var ParamValue: array of Cardinal);
```

```
void BindCardinalsByName(AnsiString ParamName,
                         Cardinal* ParamValue,
                         const int ParamValue_Size);
```

Description

Performs the same function as the BindCardinals method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindCardinalsByName(ParamName, ...) is equivalent to
BindCardinals(ParamByName(ParamName), ...).

THstmt.BindCol

```
procedure BindCol(Col: SQLUSMALLINT;
                  SqlType: SQLSMALLINT);
```

```
void BindCol(SQLUSMALLINT Col,
             SQLSMALLINT SqlType);
```

Description

This method is used to force the binding of a column Col in the result set to a given database data type SqlType. This is useful in cases where an ODBC Driver isn't able to detect the database data type of a column or where an ODBC Driver does not return the correct database data type for a column.

Note:

All BindCol calls has to be directly after Execute was called. Also, since column names are only retrieved during the binding of the columns, you cannot use the ColByName method with the BindCol method, e.g.:

BindCol(ColByName('SComment'), SQL_LONGVARCHAR);

is not allowed. ColByName will cause all the columns to bind, which makes the call to BindCol too late.

Example:

Delphi Code

```
with Hstmt1 do
begin
  SQL:= 'SELECT SNo, SName, SComment FROM Students WHERE SNo = ''9700001''';
  Prepare;
  Execute;

  //force the third column to be retrieved as a blob column
  BindCol(3, SQL_LONGVARCHAR);

  if FetchNext then
    Memo1.Text:= ColStringByName['SComment'];
end;
```

C++ Code

```
Hstmt1->SQL = "SELECT SNo, SName, SComment FROM Students WHERE SNo = '9700001'";
Hstmt1->Prepare();
Hstmt1->Execute();

//force the third column to be retrieved as a blob column
Hstmt1->BindCol(3, SQL_LONGVARCHAR);

if (Hstmt1->FetchNext()) {
  Memo1->Text = Hstmt1->ColStringByName["SComment"];
}
```

THstmt.BindDate

```
procedure BindDate(Param: SQLUSMALLINT;
                   var ParamValue: TDate);

void BindDate(SQLUSMALLINT Param,
              TDate &ParamValue);
```

Description

Used to bind an ODBC TDate parameter in a SQL statement.

BindDate(Param, ParamValue) is similar to both
BindParam(Param, SQL_C_TYPE_DATE, @ParamValue) and
BindParamCore(Param, SQL_C_TYPE_DATE, @ParamValue, SQL_TYPE_DATE)

THstmt.BindDateByName

```
procedure BindDateByName(ParamName: String;
                         var ParamValue: TDate);

void BindDateByName(AnsiString ParamName,
                    TDate &ParamValue);
```

Description

Performs the same function as the BindDate method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindDateByName(ParamName, ...) is equivalent to BindDate(ParamByName(ParamName), ...).

THstmt.BindDates

```
procedure BindDates(Param: SQLUSMALLINT;
                    var ParamValue: array of TDate);
```

```
void BindDates(SQLUSMALLINT Param,
               TDate* ParamValue,
               const int ParamValue_Size);
```

Description

Used to bind an ODBC TDate parameter array in a SQL statement.

BindDates(Param, ParamValue) is equivalent to
BindParams(Param, SQL_C_TYPE_DATE, @ParamValue, High(ParamValue)+1)

THstmt.BindDatesByName

```
procedure BindDatesByName(ParamName: String;
                          var ParamValue: array of TDate);
```

```
void BindDatesByName(AnsiString ParamName,
                     TDate* ParamValue,
                     const int ParamValue_Size);
```

Description

Performs the same function as the BindDates method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindDatesByName(ParamName, ...) is equivalent to BindDates(ParamByName(ParamName), ...).

THstmt.BindDouble

```
procedure BindDouble(Param: SQLUSMALLINT;
                     var ParamValue: Double);
```

```
void BindDouble(SQLUSMALLINT Param,
                double &ParamValue);
```

Description

Used to bind a double parameter in a SQL statement.

BindDouble(Param, ParamValue) is similar to both

BindParam(Param, SQL_C_DOUBLE, @ParamValue) and
BindParamCore(Param, SQL_C_DOUBLE, @ParamValue, SQL_DOUBLE)

THstmt.BindDoubleByName

```
procedure BindDoubleByName(ParamName: String;
                           var ParamValue: Double);
```

```
void BindDoubleByName(AnsiString ParamName,
                      double &ParamValue);
```

Description

Performs the same function as the BindDouble method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindDoubleByName(ParamName, ...) is equivalent to BindDouble(ParamByName(ParamName), ...).

THstmt.BindDoubles

```
procedure BindDoubles(Param: SQLUSMALLINT;
                      var ParamValue: array of Double);
```

```
void BindDoubles(SQLUSMALLINT Param,
                 double* ParamValue,
                 const int ParamValue_Size);
```

Description

Used to bind a double parameter array in a SQL statement.

BindDoubles(Param, ParamValue) is equivalent to
BindParams(Param, SQL_C_DOUBLE, @ParamValue, High(ParamValue)+1)

THstmt.BindDoublesByName

```
procedure BindDoublesByName(ParamName: String;
                            var ParamValue: array of Double);
```

```
void BindDoublesByName(AnsiString ParamName,
                       double* ParamValue,
                       const int ParamValue_Size);
```

Description

Performs the same function as the BindDoubles method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindDoublesByName(ParamName, ...) is equivalent to
BindDoubles(ParamByName(ParamName), ...).

THstmt.BindInt64

procedure BindInt64(Param: SQLUSMALLINT;
                        var ParamValue: **Int64**);

void BindInt64(SQLUSMALLINT Param,
                 __int64 &ParamValue);

Description

Used to bind an int64 parameter in a SQL statement.

BindInt64(Param, ParamValue) is similar to both
BindParam(Param, SQL_C_SBIGINT, @ParamValue) and
BindParamCore(Param, SQL_C_SBIGINT, @ParamValue, SQL_BIGINT)

THstmt.BindInt64ByName

procedure BindInt64ByName(ParamName: **String**;
                              var ParamValue: **Int64**);

void BindInt64ByName(**AnsiString** ParamName,
                       __int64 &ParamValue);

Description

Performs the same function as the BindInt64 method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindInt64ByName(ParamName, ...) is equivalent to BindInt64(ParamByName(ParamName), ...).

THstmt.BindInt64s

procedure BindInt64s(Param: SQLUSMALLINT;
                         var ParamValue: **array of Int64**);

void BindInt64s(SQLUSMALLINT Param,
                  __int64* ParamValue,
                  const int ParamValue_Size);

Description

Used to bind an int64 parameter array in a SQL statement.

BindInt64s(Param, ParamValue) is equivalent to
BindParams(Param, SQL_C_SBIGINT, @ParamValue, High(ParamValue)+1)

THstmt.BindInt64sByName

procedure BindInt64sByName(ParamName: **String**;

```
                                     var ParamValue: array of Int64);

void BindInt64sByName(AnsiString ParamName,
                                 __int64* ParamValue,
                                 const int ParamValue_Size);
```

Description

Performs the same function as the BindInt64s method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindInt64sByName(ParamName, ...) is equivalent to BindInt64s(ParamByName(ParamName), ...).

THstmt.BindInteger

```
procedure BindInteger(Param: SQLUSMALLINT;
                                var ParamValue: Integer);

void BindInteger(SQLUSMALLINT Param,
                       int &ParamValue);
```

Description

Used to bind an integer parameter in a SQL statement.

BindInteger(Param, ParamValue) is similar to both
BindParam(Param, SQL_C_SLONG, @ParamValue) and
BindParamCore(Param, SQL_C_SLONG, @ParamValue, SQL_INTEGER)

THstmt.BindIntegerByName

```
procedure BindIntegerByName(ParamName: String;
                                         var ParamValue: Integer);

void BindIntegerByName(AnsiString ParamName,
                                  int &ParamValue);
```

Description

Performs the same function as the BindInteger method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindIntegerByName(ParamName, ...) is equivalent to BindInteger(ParamByName(ParamName), ...).

THstmt.BindIntegers

```
procedure BindIntegers(Param: SQLUSMALLINT;
                                 var ParamValue: array of Integer);

void BindIntegers(SQLUSMALLINT Param,
```

int* ParamValue,
const int ParamValue_Size);

Description

Used to bind an integer parameter array in a SQL statement.

BindIntegers(Param, ParamValue) is equivalent to
BindParams(Param, SQL_C_SLONG, @ParamValue, High(ParamValue)+1)

THstmt.BindIntegersByName

procedure BindIntegersByName(ParamName: **String**;
var ParamValue: **array of Integer**);

void BindIntegersByName(**AnsiString** ParamName,
int* ParamValue,
const int ParamValue_Size);

Description

Performs the same function as the BindIntegers method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindIntegersByName(ParamName, ...) is equivalent to
BindIntegers(ParamByName(ParamName), ...).

THstmt.BindLongint

procedure BindLongint(Param: SQLUSMALLINT;
var ParamValue: **Longint**);

void BindLongint(SQLUSMALLINT Param,
long &ParamValue);

Description

Used to bind a longint parameter in a SQL statement.

BindLongint(Param, ParamValue) is equivalent to
BindInteger(Param, ParamValue)

THstmt.BindLongintByName

procedure BindLongintByName(ParamName: **String**;
var ParamValue: **Longint**);

void BindLongintByName(**AnsiString** ParamName,
long &ParamValue);

Description

Performs the same function as the BindLongint method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindLongintByName(ParamName, ...) is equivalent to BindLongint(ParamByName(ParamName), ...).

THstmt.BindLongints

procedure BindLongints(Param: SQLUSMALLINT;
                                    var ParamValue: **array of Longint**);

void BindLongints(SQLUSMALLINT Param,
                            long* ParamValue,
                            const int ParamValue_Size);

Description

Used to bind a longint parameter array in a SQL statement.

BindLongints(Param, ParamValue) is equivalent to
BindIntegers(Param, ParamValue)

THstmt.BindLongintsByName

procedure BindLongintsByName(ParamName: **String**;
                                                var ParamValue: **array of Longint**);

void BindLongintsByName(**AnsiString** ParamName,
                                        long* ParamValue,
                                        const int ParamValue_Size);

Description

Performs the same function as the BindLongints method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindLongintsByName(ParamName, ...) is equivalent to
BindLongints(ParamByName(ParamName), ...).

THstmt.BindLongword

procedure BindLongword(Param: SQLUSMALLINT;
                                      var ParamValue: **Longword**);

void BindLongword(SQLUSMALLINT Param,
                              Longword &ParamValue);

Description

Used to bind a longword parameter in a SQL statement.

BindLongword(Param, ParamValue) is equivalent to

BindCardinal(Param, ParamValue)

THstmt.BindLongwordByName

procedure BindLongwordByName(ParamName: **String**;
                                                  var ParamValue: **Longword**);

void BindLongwordByName(**AnsiString** ParamName,
                                        Longword &ParamValue);

Description

Performs the same function as the BindLongword method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindLongwordByName(ParamName, ...) is equivalent to
BindLongword(ParamByName(ParamName), ...).

THstmt.BindLongwords

procedure BindLongwords(Param: SQLUSMALLINT;
                                       var ParamValue: **array of Longword**);

void BindLongwords(SQLUSMALLINT Param,
                               Longword* ParamValue,
                               const int ParamValue_Size);

Description

Used to bind a longword parameter array in a SQL statement.

BindLongwords(Param, ParamValue) is equivalent to
BindCardinals(Param, ParamValue)

THstmt.BindLongwordsByName

procedure BindLongwordsByName(ParamName: **String**;
                                                    var ParamValue: **array of Longword**);

void BindLongwordsByName(**AnsiString** ParamName,
                                          Longword* ParamValue,
                                          const int ParamValue_Size);

Description

Performs the same function as the BindLongwords method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindLongwordsByName(ParamName, ...) is equivalent to
BindLongwords(ParamByName(ParamName), ...).

THstmt.BindMemory

```
procedure BindMemory(Param: SQLUSMALLINT;
                     var ParamValue: TMemoryStream;
                     Binary: Boolean);
```

```
void BindMemory(SQLUSMALLINT Param,
                TMemoryStream* &ParamValue,
                bool Binary);
```

Description

Used to bind a blob parameter in a SQL statement. The Binary parameter specifies whether the blob must be stored as binary data or as text data. This method also automatically sets the ParamSize property using the ParamValue.Size property.

BindMemory(Param, ParamValue, True) is similar to both
BindParam(Param, SQL_C_BINARY, @ParamValue) and
BindParamCore(Param, SQL_C_BINARY, @ParamValue, SQL_LONGVARBINARY)

BindMemory(Param, ParamValue, False) is similar to both
BindParam(Param, SQL_C_CHAR, @ParamValue) and
BindParamCore(Param, SQL_C_CHAR, @ParamValue, SQL_LONGVARCHAR)

THstmt.BindMemoryByName

```
procedure BindMemoryByName(ParamName: String;
                           var ParamValue: TMemoryStream;
                           Binary: Boolean);
```

```
void BindMemoryByName(AnsiString ParamName,
                      TMemoryStream* &ParamValue,
                      bool Binary);
```

Description

Performs the same function as the BindMemory method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindMemoryByName(ParamName, ...) is equivalent to
BindMemory(ParamByName(ParamName), ...).

THstmt.BindNull

```
procedure BindNull(Param: SQLUSMALLINT);
```

```
void BindNull(SQLUSMALLINT Param);
```

Description

Used to bind a null value for a parameter in a SQL statement.

BindNull(Param) is similar to both

BindParam(Param, SQL_C_DEFAULT, nil) and
BindParamCore(Param, SQL_C_DEFAULT, nil, SQL_CHAR)

THstmt.BindNullByName

procedure BindNullByName(ParamName: **String**);

void BindNullByName(**AnsiString** ParamName);

Description

Performs the same function as the BindNull method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindNullByName(ParamName) is equivalent to BindNull(ParamByName(ParamName)).

THstmt.BindNulls

procedure BindNulls(Param: SQLUSMALLINT);

void BindNulls(SQLUSMALLINT Param);

Description

Used to bind a null value parameter array in a SQL statement.

BindNulls(Param) is equivalent to
BindParams(Param, SQL_C_DEFAULT, nil, BulkSize)

Note:

The BulkSize property has to be set before this method can be called. In other words this method can be called either after the BulkSize property was set in the code or after a call to one of the other bulk bind methods was made.

THstmt.BindNullsByName

procedure BindNullsByName(ParamName: **String**);

void BindNullsByName(**AnsiString** ParamName);

Description

Performs the same function as the BindNullStrings method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindNullsByName(ParamName) is equivalent to BindNulls(ParamByName(ParamName)).

THstmt.BindNullString

```
procedure BindNullString(Param: SQLUSMALLINT;
                         var ParamValue: NullString);
```

```
void BindNullString(SQLUSMALLINT Param,
                    NullString &ParamValue);
```

Description

Used to bind a NullString parameter in a SQL statement.

BindNullString(Param, ParamValue) is similar to both
BindParam(Param, SQL_C_CHAR, @ParamValue) and
BindParamCore(Param, SQL_C_CHAR, @ParamValue, SQL_CHAR)

THstmt.BindNullStringByName

```
procedure BindNullStringByName(ParamName: String;
                               var ParamValue: NullString);
```

```
void BindNullStringByName(AnsiString ParamName,
                          NullString &ParamValue);
```

Description

Performs the same function as the BindNullString method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindNullStringByName(ParamName, ...) is equivalent to
BindNullString(ParamByName(ParamName), ...).

THstmt.BindNullStrings

```
procedure BindNullStrings(Param: SQLUSMALLINT;
                          var ParamValue: array of NullString);
```

```
void BindNullStrings(SQLUSMALLINT Param,
                     NullString* ParamValue,
                     const int ParamValue_Size);
```

Description

Used to bind a NullString parameter array in a SQL statement.

BindNullStrings(Param, ParamValue) is equivalent to
BindParams(Param, SQL_C_CHAR, @ParamValue, High(ParamValue)+1)

THstmt.BindNullStringsByName

```
procedure BindNullStringsByName(ParamName: String;
                                var ParamValue: array of NullString);
```

```
void BindNullStringsByName(AnsiString ParamName,
                              NullString* ParamValue,
                              const int ParamValue_Size);
```

Description

Performs the same function as the BindNullStrings method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindNullStringsByName(ParamName, ...) is equivalent to
BindNullStrings(ParamByName(ParamName), ...).

THstmt.BindParam

```
procedure BindParam(Param: SQLUSMALLINT;
                    ParamType: SQLSMALLINT;
                    ParamValue: SQLPOINTER);

procedure BindParam(Param: SQLUSMALLINT;
                    ParamType: SQLSMALLINT;
                    ParamValue: SQLPOINTER;
                    SqlType: SQLSMALLINT);

void BindParam(SQLUSMALLINT Param,
               SQLSMALLINT ParamType,
               SQLPOINTER ParamValue);

void BindParam(SQLUSMALLINT Param,
               SQLSMALLINT ParamType,
               SQLPOINTER ParamValue,
               SQLSMALLINT SqlType);
```

Description

Used to supply the storage for a named or unnamed parameter in a SQL statement. For each parameter number Param (1 based), you must bind in the storage type ParamType and the pointer to the parameter value ParamValue. If the parameter value bound is of a blob type, you must also set the ParamSize property.

The overloaded BindParam method with the additional parameter SqlType allows ODBCExpress to fall back on using the SqlType parameter when the more accurate SQLDescribeParam function, which return this and other values, fails. The other values mentioned are then supplemented by the driver profiles. Some ODBC drivers, which support the SQLDescribeParam function, fail when the query is too complex (sub-selects, unions, etc), preventing you from using certain dynamic parameters in this query. This BindParam method solves this problem, and should therefore be used in favour of the other BindParam method and the BindParamCore method (unless you are using these two methods to force the usage or non-usage of the SQLDescribeParam function).

Example:

Delphi Code

```
var
  SNo: NullString;
```

```
begin
  with Hstmt1 do
  begin
    //set and prepare a select statement with a parameter
    SQL:= 'SELECT SNo, SName FROM Students WHERE SNo LIKE ?';
    Prepare;

    //bind the parameter using the BindParam method
    BindParam(1, SQL_C_CHAR, @SNo);

    //or

    //bind the parameter using the other overloaded BindParam method
    BindParam(1, SQL_C_CHAR, @SNo, SQL_CHAR);

    //or

    //bind the parameter using the simpler BindNullString method
    BindNullString(1, SNo);

    //assign the parameter value and execute the select statement
    SNo:= '97%';
    Execute;
  end;
end;
```

C++ Code

```
NullString SNo;

//set and prepare a select statement with a parameter
Hstmt1->SQL = "SELECT SNo, SName FROM Students WHERE SNo LIKE ?";
Hstmt1->Prepare();

//bind the parameter using the BindParam method
Hstmt1->BindParam(1, SQL_C_CHAR, &SNo);

//or

//bind the parameter using the other overloaded BindParam method
Hstmt1->BindParam(1, SQL_C_CHAR, &SNo, SQL_CHAR);

//or

//bind the parameter using the simpler BindNullString method
Hstmt1->BindNullString(1, SNo);

//assign the parameter value and execute the select statement
strcpy(SNo, "97%");
Hstmt1->Execute();
```

THstmt.BindParamCore

procedure BindParamCore(Param: SQLUSMALLINT;

```
                          ParamType: SQLSMALLINT;
                          ParamValue: SQLPOINTER;
                          SqlType: SQLSMALLINT);
```

```
void BindParamCore(SQLUSMALLINT Param,
                   SQLSMALLINT ParamType,
                   SQLPOINTER ParamValue,
                   SQLSMALLINT SqlType);
```

Description

An alternative to the BindParam method. BindParam makes use of level 2 ODBC functionality to determine the database data types of parameters in the prepared SQL statement. BindParamCore works similarly to BindParam, except that it does not automatically detect the database data types of the parameters in the SQL statement. You must therefore supply the database data type SqlType for each parameter that you bind. BindParamCore doesn't make use of level 2 ODBC functionality.

Example:

Delphi Code

```
var
  SNo: NullString;
begin
  with Hstmt1 do
  begin
    //set and prepare a select statement with a parameter
    SQL:= 'SELECT SNo, SName FROM Students WHERE SNo LIKE ?';
    Prepare;

    //bind the parameter using the BindParamCore method
    BindParamCore(1, SQL_C_CHAR, @SNo, SQL_CHAR);

    //or

    //bind the parameter using the simpler BindNullString method
    BindNullString(1, SNo);

    //assign the parameter value and execute the select statement
    SNo:= '97%';
    Execute;
  end;
end;
```

C++ Code

```
NullString SNo;

//set and prepare a select statement with a parameter
Hstmt1->SQL = "SELECT SNo, SName FROM Students WHERE SNo LIKE ?";
Hstmt1->Prepare();

//bind the parameter using the BindParam method
Hstmt1->BindParamCore(1, SQL_C_CHAR, &SNo, SQL_CHAR);
```

```
//or

//bind the parameter using the simpler BindNullString method
Hstmt1->BindNullString(1, SNo);

//assign the parameter value and execute the select statement
strcpy(SNo, "97%");
Hstmt1->Execute();
```

THstmt.BindParams

```
procedure BindParams(Param: SQLUSMALLINT;
                     ParamType: SQLSMALLINT;
                     ParamValue: SQLPOINTER;
                     Bulk: Integer);

void BindParams(SQLUSMALLINT Param,
                SQLSMALLINT ParamType,
                SQLPOINTER ParamValue,
                int Bulk);
```

Description

Used to supply an array of storage for a named or unnamed parameter in a SQL statement. For each parameter number Param (1 based), you must bind in the storage type ParamType and the pointer to the parameter array ParamValue. This method is used to bind in multiple values for the same parameter, as opposed to the BindParam method which only allows you to bind in one value per parameter. It is used to perform bulk operations at a DataSource such as bulk inserts. Bulk indicates the size of the array that is being bound. If the number of rows bound is less than the size of the array, then the BulkSize property must be set after all the arrays were bound.

Example:

Delphi Code

```
var
  //declare arrays to hold up to a 1000 rows each
  SNo: array[1..1000] of NullString;
  SGender: array[1..1000] of Byte;
begin
  with Hstmt1 do
  begin
    //set and prepare a SQL statement to do a bulk insert
    SQL:= 'INSERT INTO Students (SNo, SGender) VALUES (?, ?)';
    Prepare;

    //bind the parameter arrays using the BindParams method
    BindParams(1, SQL_C_CHAR, @SNo, High(SNo)+1);
    BindParams(1, SQL_C_UTINYINT, @SGender, High(SGender)+1);

    //or

    //bind the parameter arrays using the simpler methods provided
    BindNullStrings(1, SNo);
    BindBytes(2, SGender);

    //assign values to the bulk arrays and set the bulk size if number of rows < 1000
    BulkSize:= 900;

    //insert the rows into the database
    Execute;
  end;
end;
```

C++ Code

```
//declare arrays to hold up to a 1000 rows each
NullString SNo[1000];
Byte SGender[1000];

//set and prepare a SQL statement to do a bulk insert
Hstmt1->SQL = "INSERT INTO Students (SNo, SGender) VALUES (?, ?)";
Hstmt1->Prepare();

//bind the parameter arrays using the BindParams method
Hstmt1->BindParams(1, SQL_C_CHAR, &SNo, 1000);
Hstmt1->BindParams(1, SQL_C_UTINYINT, &SGender, 1000);

//or

//bind the parameter arrays using the simpler methods provided
Hstmt1->BindNullStrings(1, SNo, 1000);
Hstmt1->BindBytes(2, SGender, 1000);

//assign values to the bulk arrays and set the bulk size if number of rows < 1000
Hstmt1->BulkSize = 900;

//insert the rows into the database
Hstmt1->Execute();
```

THstmt.BindShortint

```
procedure BindShortint(Param: SQLUSMALLINT;
                                    var ParamValue: Shortint);
```

```
void BindShortint(SQLUSMALLINT Param,
                           Shortint &ParamValue);
```

Description

Used to bind a shortint parameter in a SQL statement.

BindShortint(Param, ParamValue) is similar to both
BindParam(Param, SQL_C_STINYINT, @ParamValue) and
BindParamCore(Param, SQL_C_STINYINT, @ParamValue, SQL_TINYINT)

THstmt.BindShortintByName

```
procedure BindShortintByName(ParamName: String;
                             var ParamValue: Shortint);
```

```
void BindShortintByName(AnsiString ParamName,
                        Shortint &ParamValue);
```

Description

Performs the same function as the BindString method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindShortintByName(ParamName, ...) is equivalent to
BindShortint(ParamByName(ParamName), ...).

THstmt.BindShortints

```
procedure BindShortints(Param: SQLUSMALLINT;
                        var ParamValue: array of Shortint);
```

```
void BindShortints(SQLUSMALLINT Param,
                   Shortint* ParamValue,
                   const int ParamValue_Size);
```

Description

Used to bind a shortint parameter array in a SQL statement.

BindShortints(Param, ParamValue) is equivalent to
BindParams(Param, SQL_C_STINYINT, @ParamValue, High(ParamValue)+1)

THstmt.BindShortintsByName

```
procedure BindShortintsByName(ParamName: String;
                              var ParamValue: array of Shortint);
```

```
void BindShortintsByName(AnsiString ParamName,
                         Shortint* ParamValue,
                         const int ParamValue_Size);
```

Description

Performs the same function as the BindShortints method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindShortintsByName(ParamName, ...) is equivalent to
BindShortints(ParamByName(ParamName), ...).

THstmt.BindSingle

procedure BindSingle(Param: SQLUSMALLINT;
                            var ParamValue: **Single**);

void BindSingle(SQLUSMALLINT Param,
                    float &ParamValue);

Description

Used to bind a single parameter in a SQL statement.

BindSingle(Param, ParamValue) is similar to both
BindParam(Param, SQL_C_FLOAT, @ParamValue) and
BindParamCore(Param, SQL_C_FLOAT, @ParamValue, SQL_REAL)

THstmt.BindSingleByName

procedure BindSingleByName(ParamName: **String**;
                                    var ParamValue: **Single**);

void BindSingleByName(**AnsiString** ParamName,
                            float &ParamValue);

Description

Performs the same function as the BindSingle method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindSingleByName(ParamName, ...) is equivalent to BindSingle(ParamByName(ParamName), ...).

THstmt.BindSingles

procedure BindSingles(Param: SQLUSMALLINT;
                            var ParamValue: **array of Single**);

void BindSingles(SQLUSMALLINT Param,
                    float* ParamValue,
                    const int ParamValue_Size);

Description

Used to bind a single parameter array in a SQL statement.

BindSingles(Param, ParamValue) is equivalent to
BindParams(Param, SQL_C_FLOAT, @ParamValue, High(ParamValue)+1)

THstmt.BindSinglesByName

procedure BindSinglesByName(ParamName: **String**;
                                    var ParamValue: **array of Single**);

void BindSinglesByName(**AnsiString** ParamName,

```
                              float* ParamValue,
                              const int ParamValue_Size);
```

Description

Performs the same function as the BindSingles method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindSinglesByName(ParamName, ...) is equivalent to BindSingles(ParamByName(ParamName), ...).

THstmt.BindSmallint

```
procedure BindSmallint(Param: SQLUSMALLINT;
                       var ParamValue: Smallint);

void BindSmallint(SQLUSMALLINT Param,
                  short &ParamValue);
```

Description

Used to bind a smallint parameter in a SQL statement.

BindSmallint(Param, ParamValue) is similar to both
BindParam(Param, SQL_C_SSHORT, @ParamValue) and
BindParamCore(Param, SQL_C_SSHORT, @ParamValue, SQL_SMALLINT)

THstmt.BindSmallintByName

```
procedure BindSmallintByName(ParamName: String;
                             var ParamValue: Smallint);

void BindSmallintByName(AnsiString ParamName,
                        short &ParamValue);
```

Description

Performs the same function as the BindSmallint method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindSmallintByName(ParamName, ...) is equivalent to
BindSmallint(ParamByName(ParamName), ...).

THstmt.BindSmallints

```
procedure BindSmallints(Param: SQLUSMALLINT;
                        var ParamValue: array of Smallint);

void BindSmallints(SQLUSMALLINT Param,
                   short* ParamValue,
                   const int ParamValue_Size);
```

Description

Used to bind a smallint parameter array in a SQL statement.

BindSmallints(Param, ParamValue) is equivalent to
BindParams(Param, SQL_C_SSHORT, @ParamValue, High(ParamValue)+1)

THstmt.BindSmallintsByName

```
procedure BindSmallintsByName(ParamName: String;
                              var ParamValue: array of Smallint);
```

```
void BindSmallintsByName(AnsiString ParamName,
                         short* ParamValue,
                         const int ParamValue_Size);
```

Description

Performs the same function as the BindSmallints method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindSmallintsByName(ParamName, ...) is equivalent to
BindSmallints(ParamByName(ParamName), ...).

THstmt.BindString

```
procedure BindString(Param: SQLUSMALLINT;
                     var ParamValue: String);
```

```
void BindString(SQLUSMALLINT Param,
                AnsiString &ParamValue);
```

Description

Used to bind a string parameter in a SQL statement.

BindString(Param, ParamValue) is similar to both
BindParam(Param, SQL_C_CHAR, PChar(ParamValue)) and
BindParamCore(Param, SQL_C_CHAR, PChar(ParamValue), SQL_CHAR)

Note:

If this method is used to bind memo (string blob) parameters, you might have to set the ParamSize property to the size of the memo before executing the statement. This is only necessary for some ODBC drivers, if you find that all the memo text is not transferred to the database.

THstmt.BindStringByName

```
procedure BindStringByName(ParamName: String;
                           var ParamValue: String);
```

```
void BindStringByName(AnsiString ParamName,
                      AnsiString &ParamValue);
```

Description

Performs the same function as the BindString method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindStringByName(ParamName, ...) is equivalent to BindString(ParamByName(ParamName), ...).

THstmt.BindText

```
procedure BindText(Param: SQLUSMALLINT;
                   var ParamValue: TMemoryStream);
```

```
void BindText(SQLUSMALLINT Param,
              TMemoryStream* &ParamValue);
```

Description

Used to bind a text blob parameter in a SQL statement.

BindText(Param, ParamValue) is equivalent to
BindMemory(Param, ParamValue, False)

THstmt.BindTextByName

```
procedure BindTextByName(ParamName: String;
                         var ParamValue: TMemoryStream);
```

```
void BindTextByName(AnsiString ParamName,
                    TMemoryStream* &ParamValue);
```

Description

Performs the same function as the BindText method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindTextByName(ParamName, ...) is equivalent to BindText(ParamByName(ParamName), ...).

THstmt.BindTime

```
procedure BindTime(Param: SQLUSMALLINT;
                   var ParamValue: TTime);
```

```
void BindTime(SQLUSMALLINT Param,
              TTime &ParamValue);
```

Description

Used to bind an ODBC TTime parameter in a SQL statement.

BindTime(Param, ParamValue) is similar to both
BindParam(Param, SQL_C_TYPE_TIME, @ParamValue) and
BindParamCore(Param, SQL_C_TYPE_TIME, @ParamValue, SQL_TYPE_TIME)

THstmt.BindTimeByName

```
procedure BindTimeByName(ParamName: String;
                         var ParamValue: TTime);
```

```
void BindTimeByName(AnsiString ParamName,
                    TTime &ParamValue);
```

Description

Performs the same function as the BindTime method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindTimeByName(ParamName, ...) is equivalent to BindTime(ParamByName(ParamName), ...).

THstmt.BindTimes

```
procedure BindTimes(Param: SQLUSMALLINT;
                    var ParamValue: array of TTime);
```

```
void BindTimes(SQLUSMALLINT Param,
               TTime* ParamValue,
               const int ParamValue_Size);
```

Description

Used to bind an ODBC TTime parameter array in a SQL statement.

BindTimes(Param, ParamValue) is equivalent to
BindParams(Param, SQL_C_TYPE_TIME, @ParamValue, High(ParamValue)+1)

THstmt.BindTimesByName

```
procedure BindTimesByName(ParamName: String;
                          var ParamValue: array of TTime);
```

```
void BindTimesByName(AnsiString ParamName,
                     TTime* ParamValue,
                     const int ParamValue_Size);
```

Description

Performs the same function as the BindTimes method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindTimesByName(ParamName, ...) is equivalent to BindTimes(ParamByName(ParamName), ...).

THstmt.BindTimeStamp

```
procedure BindTimeStamp(Param: SQLUSMALLINT;
                        var ParamValue: TTimeStamp);

void BindTimeStamp(SQLUSMALLINT Param,
                   TTimeStamp &ParamValue);
```

Description

Used to bind an ODBC TTimeStamp parameter in a SQL statement.

BindTimeStamp(Param, ParamValue) is similar to both BindParam(Param, SQL_C_TYPE_TIMESTAMP, @ParamValue) and BindParamCore(Param, SQL_C_TYPE_TIMESTAMP, @ParamValue, SQL_TYPE_TIMESTAMP)

THstmt.BindTimeStampByName

```
procedure BindTimeStampByName(ParamName: String;
                              var ParamValue: TTimeStamp);

void BindTimeStampByName(AnsiString ParamName,
                         TTimeStamp &ParamValue);
```

Description

Performs the same function as the BindTimeStamp method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindTimeStampByName(ParamName, ...) is equivalent to BindTimeStamp(ParamByName(ParamName), ...).

THstmt.BindTimeStamps

```
procedure BindTimeStamps(Param: SQLUSMALLINT;
                         var ParamValue: array of TTimeStamp);

void BindTimeStamps(SQLUSMALLINT Param,
                    TTimeStamp* ParamValue,
                    const int ParamValue_Size);
```

Description

Used to bind an ODBC TTimeStamp parameter array in a SQL statement.

BindTimeStamps(Param, ParamValue) is equivalent to BindParams(Param, SQL_C_TYPE_TIMESTAMP, @ParamValue, High(ParamValue)+1)

THstmt.BindTimeStampsByName

procedure BindTimeStampsByName(ParamName: **String**;
var ParamValue: **array of** TTimeStamp);

void BindTimeStampsByName(**AnsiString** ParamName,
TTimeStamp* ParamValue,
const int ParamValue_Size);

Description

Performs the same function as the BindTimeStamps method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindTimeStampsByName(ParamName, ...) is equivalent to
BindTimeStamps(ParamByName(ParamName), ...).

THstmt.BindWord

procedure BindWord(Param: SQLUSMALLINT;
var ParamValue: **Word**);

void BindWord(SQLUSMALLINT Param,
Word &ParamValue);

Description

Used to bind a word parameter in a SQL statement.

BindWord(Param, ParamValue) is similar to both
BindParam(Param, SQL_C_USHORT, @ParamValue) and
BindParamCore(Param, SQL_C_USHORT, @ParamValue, SQL_SMALLINT)

THstmt.BindWordByName

procedure BindWordByName(ParamName: **String**;
var ParamValue: **Word**);

void BindWordByName(**AnsiString** ParamName,
Word &ParamValue);

Description

Performs the same function as the BindWord method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindWordByName(ParamName, ...) is equivalent to BindWord(ParamByName(ParamName), ...).

THstmt.BindWords

procedure BindWords(Param: SQLUSMALLINT;

var ParamValue: **array of Word**);

void BindWords(SQLUSMALLINT Param,
Word* ParamValue,
const int ParamValue_Size);

Description

Used to bind a word parameter array in a SQL statement.

BindWords(Param, ParamValue) is equivalent to
BindParams(Param, SQL_C_USHORT, @ParamValue, High(ParamValue)+1)

THstmt.BindWordsByName

procedure BindWordsByName(ParamName: **String**;
var ParamValue: **array of Word**);

void BindWordsByName(**AnsiString** ParamName,
Word* ParamValue,
const int ParamValue_Size);

Description

Performs the same function as the BindWords method, except that it takes a parameter name instead of a parameter number to indicate a parameter in the SQL statement.

BindWordsByName(ParamName, ...) is equivalent to BindWords(ParamByName(ParamName), ...).

THstmt.BlobCol

property BlobCol[Col: SQLUSMALLINT]: **Boolean**;

property bool BlobCol[SQLUSMALLINT Col];

Description

It is used to determine if a column in the result set is bound or not. If BlobCol returns True, then the ColMemory and CellMemory properties will return a valid memory stream which contains the column data for column Col of the current row in the result set.

THstmt.BlobDeferral

property BlobDeferral: **Boolean**;

property bool BlobDeferral;

Description

If this property is True, each blob column of the current fetched row will only be retrieved from the DataSource when (if ever) the column is referenced, e.g. ColMemory[3] will automatically fetch the blob of column 3 from the DataSource if not fetched already. The default value is False. When you retrieve multiple rows on each fetch (RowSetSize > 1), you can include blob columns. In this case blob deferral will automatically be applied, while a blob is only fetched when a blob cell of the fetched rows is referenced, e.g. CellMemory[3,4] will automatically retrieve the blob of column 3 of row 4 in the current row set.

THstmt.BlobPlacement

TBlobPlacement = (bpDetect, bpByParts, bpByExec);

property BlobPlacement: TBlobPlacement;

TBlobPlacement {bpDetect, bpByParts, bpByExec};

property TBlobPlacement BlobPlacement;

Description

This property can be used to change the way used by the statement component to place blobs in a database. Since in most cases the component detects which is the correct method to use, this property should be left at the default value if possible. Only in certain cases where the method detected does not work well with the ODBC Driver you are using, then you should change it to one of its other two values, as described below. The property can have one of the following values:

bpDetect	Detect which method to use. This is the default value.
bpByParts	Place blobs in the database by parts after execution of the SQL statement.
bpByExec	Place blobs in the database during execution of the SQL statement.

Note:

Most ODBC Drivers only supports one of the two methods, so it should always be left up to the statement component to detect which method to use, unless it's not possible.

THstmt.BlobSize

property BlobSize: **Longint**;

property long BlobSize;

Description

Used to specify the smallest amount of memory to be allocated when the first part of a blob is retrieved. For most ODBC Drivers, this value can be left unchanged, since the memory allocated for a blob will be dynamically grown when a blob is retrieved. However, some ODBC Drivers are unable to return the total size of a blob before the blob is retrieved. For these drivers, the BlobSize property has to be set to at least the size of the current blob retrieved. A general way of doing this is to determine the size of the largest blob in the table and set the BlobSize property to that size.

THstmt.Bookmark

property Bookmark: SQLPOINTER;

property SQLPOINTER Bookmark;

Description

Returns a pointer to a variable length bookmark for the current row in the result set. This bookmark value can later be used together with the FetchBookmark method to return to the current row in the result set. When not needed anymore, this bookmark must be freed using the FreeBookmark method.

THstmt.BookmarkSize

property BookmarkSize: SQLINTEGER;

property SQLINTEGER BookmarkSize;

Description

Returns the physical size of the bookmark values for the current result set. Pointers to bookmark values can be obtained using the Bookmark property.

THstmt.BulkData

property BulkData: **Boolean**;

property bool BulkData;

Description

When set to True, this property will cause string columns to be retrieved as standard 256 byte length strings, which can then be directly bound to another THstmt component as bulk parameters. The default value is False.

Example:

Delphi Code

```
//if BulkData is set to True on HstmtSelect
//then # may indicate a string parameter and column
HstmtInsert.BindParams(#, HstmtSelect.ColType[#],
                          HstmtSelect.ColValue[#],
                          HstmtSelect.RowSetSize);
```

C++ Code

```
//if BulkData is set to True on HstmtSelect
//then # may indicate a string parameter and column
HstmtInsert->BindParams(#, HstmtSelect.ColType[#],
                           HstmtSelect.ColValue[#],
                           HstmtSelect.RowSetSize);
```

Setting this property to True will also allow you to bind null values for blob data in bulk, since normally you are not allowed to bind blob columns in bulk under any circumstance.

Example:

Delphi Code

```
//if BulkData is set to True on HstmtInsert
//then # may indicate a string parameter and column
if HstmtSelect.BlobCol[#] then
  HstmtInsert.BindNulls(#)
else
  HstmtInsert.BindParams(#, ...);
```

C++ Code

```
//if BulkData is set to True on HstmtInsert
//then # may indicate a string parameter and column
if (HstmtSelect->BlobCol[#]) {
  HstmtInsert->BindNulls(#);
}
else {
  HstmtInsert->BindParams(#, ...);
}
```

THstmt.BulkParamSize

```
property BulkParamSize[Param: SQLUSMALLINT;
                       Row: SQLUINTEGER]: SQLINTEGER;
```

```
property SQLINTEGER BulkParamSize[SQLUSMALLINT Param][SQLUINTEGER Row];
```

Description

This property can be used to read or set the data size of row number Row of bulk parameter number Param. The data size of each row of a bulk parameter is automatically set when binding the bulk parameter.

THstmt.BulkSize

```
property BulkSize: SQLUINTEGER;
```

```
property SQLUINTEGER BulkSize;
```

Description

Used to indicate the number of rows bound with the BindParams methods. This property must be set after all the parameter arrays were bound. If the number of rows bound is one greater than the high value of the largest array bound (High(LargestArray)+1), then it's not necessary to set this property. If the arrays bound are of different sizes, then the BulkSize property must be set to a value less than or equal to the number of elements that the smallest array can hold (High(SmallestArray)+1).

Example:

Delphi Code

```
var
  //declare arrays to hold up to a 1000 rows each
  SNo: array[1..1000] of NullString;
  SGender: array[1..1000] of Byte;
begin
  with Hstmt1 do
  begin
    //set and prepare a SQL statement to do a bulk insert
    SQL:= 'INSERT INTO Students (SNo, SGender) VALUES (?, ?)';
    Prepare;

    //bind the bulk arrays
    BindNullStrings(1, SNo);
    BindBytes(2, SGender);

    //assign values to the bulk arrays and set the bulk size if number of rows < 1000
    BulkSize:= 900;

    //insert the rows into the database
    Execute;
  end;
end;
```

C++ Code

```
//declare arrays to hold up to a 1000 rows each
NullString SNo[1000];
Byte SGender[1000];

//set and prepare a SQL statement to do a bulk insert
Hstmt1->SQL = "INSERT INTO Students (SNo, SGender) VALUES (?, ?)";
Hstmt1->Prepare();

//bind the bulk arrays
Hstmt1->BindNullStrings(1, SNo, 1000);
Hstmt1->BindBytes(2, SGender, 1000);

//assign values to the bulk arrays and set the bulk size if number of rows < 1000
Hstmt1->BulkSize = 900;

//insert the rows into the database
Hstmt1->Execute();
```

THstmt.CellBoolean

property CellBoolean[Col, Row: SQLUSMALLINT]: **Boolean**;

property bool CellBoolean[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the CellString property, except that it returns a boolean.

THstmt.CellBooleanByName

property CellBooleanByName[ColName: **String**;
Row: SQLUSMALLINT]: **Boolean**;

property bool CellBooleanByName[**AnsiString** ColName][SQLUSMALLINT Row];

Description

Performs the same function as the CellBoolean property, except that it takes a column name instead of a column number to indicate a column in the result set.

CellBooleanByName[ColName] is equivalent to CellBoolean[ColByName(ColName)].

THstmt.CellByte

property CellByte[Col, Row: SQLUSMALLINT]: **Byte**;

property Byte CellByte[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the CellString property, except that it returns a byte.

THstmt.CellByteByName

property CellByteByName[ColName: **String**;
Row: SQLUSMALLINT]: **Byte**;

property Byte CellByteByName[**AnsiString** ColName][SQLUSMALLINT Row];

Description

Performs the same function as the CellByte property, except that it takes a column name instead of a column number to indicate a column in the result set.

CellByteByName[ColName] is equivalent to CellByte[ColByName(ColName)].

THstmt.CellCardinal

property CellCardinal[Col, Row: SQLUSMALLINT]: **Cardinal**;

property Cardinal CellCardinal[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the CellString property, except that it returns a cardinal.

THstmt.CellCardinalByName

property CellCardinalByName[ColName: **String**;
Row: SQLUSMALLINT]: **Cardinal**;

property Cardinal CellCardinalByName[**AnsiString** ColName][SQLUSMALLINT Row];

Description

Performs the same function as the CellCardinal property, except that it takes a column name instead of a column number to indicate a column in the result set.

CellCardinalByName[ColName] is equivalent to CellCardinal[ColByName(ColName)].

THstmt.CellDate

property CellDate[Col, Row: SQLUSMALLINT]: TDate;

property TDate CellDate[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the CellString property, except that it returns an ODBC TDate.

THstmt.CellDateByName

property CellDateByName[ColName: **String**;
Row: SQLUSMALLINT]: TDate;

property TDate CellDateByName[**AnsiString** ColName][SQLUSMALLINT Row];

Description

Performs the same function as the CellDate property, except that it takes a column name instead of a column number to indicate a column in the result set.

CellDateByName[ColName] is equivalent to CellDate[ColByName(ColName)].

THstmt.CellDouble

property CellDouble[Col, Row: SQLUSMALLINT]: **Double**;

property double CellDouble[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the CellString property, except that it returns a double.

THstmt.CellDoubleByName

property CellDoubleByName[ColName: **String**;
Row: SQLUSMALLINT]: **Double**;

property double CellDoubleByName[**AnsiString** ColName][SQLUSMALLINT Row];

Description

Performs the same function as the CellDouble property, except that it takes a column name instead of a column number to indicate a column in the result set.

CellDoubleByName[ColName] is equivalent to CellDouble[ColByName(ColName)].

THstmt.CellIgnore

property CellIgnore[Col, Row: SQLUSMALLINT]: **Boolean**;

property bool CellIgnore[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the ColIgnore property, except that it applies to row number Row in the current row set of the current result set.

THstmt.CellInt64

property CellInt64[Col, Row: SQLUSMALLINT]: **Int64**;

property __int64 CellInt64[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the CellString property, except that it returns an int64.

THstmt.CellInt64ByName

property CellInt64ByName[ColName: **String**;
Row: SQLUSMALLINT]: **Int64**;

property __int64 CellInt64ByName[**AnsiString** ColName][SQLUSMALLINT Row];

Description

Performs the same function as the CellInt64 property, except that it takes a column name instead of a column number to indicate a column in the result set.

CellInt64ByName[ColName] is equivalent to CellInt64[ColByName(ColName)].

THstmt.CellInteger

property CellInteger[Col, Row: SQLUSMALLINT]: **Integer**;

property int CellInteger[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the CellString property, except that it returns an integer.

THstmt.CellIntegerByName

property CellIntegerByName[ColName: **String**;
Row: SQLUSMALLINT]: **Integer**;

property int CellIntegerByName[**AnsiString** ColName][SQLUSMALLINT Row];

Description

Performs the same function as the CellInteger property, except that it takes a column name instead of a column number to indicate a column in the result set.

CellIntegerByName[ColName] is equivalent to CellInteger[ColByName(ColName)].

THstmt.CellLongint

property CellLongint[Col, Row: SQLUSMALLINT]: **Longint**;

property long CellLongint[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the CellString property, except that it returns a longint.

THstmt.CellLongintByName

property CellLongintByName[ColName: **String**;
Row: SQLUSMALLINT]: **Longint**;

property long CellLongintByName[**AnsiString** ColName][SQLUSMALLINT Row];

Description

Performs the same function as the CellLongint property, except that it takes a column name instead of a column number to indicate a column in the result set.

CellLongintByName[ColName] is equivalent to CellLongint[ColByName(ColName)].

THstmt.CellLongword

property CellLongword[Col, Row: SQLUSMALLINT]: **Longword**;

property Longword CellLongword[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the CellString property, except that it returns a longword.

THstmt.CellLongwordByName

property CellLongwordByName[ColName: **String**;
Row: SQLUSMALLINT]: **Longword**;

property Longword CellLongwordByName[**AnsiString** ColName][SQLUSMALLINT Row];

Description

Performs the same function as the CellLongword property, except that it takes a column name instead of a column number to indicate a column in the result set.

CellLongwordByName[ColName] is equivalent to CellLongword[ColByName(ColName)].

THstmt.CellMemory

property CellMemory[Col, Row: SQLUSMALLINT]: TMemoryStream;

property TMemoryStream* CellMemory[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the CellString property, except that it returns a memory stream.

Note:

When referencing this property, the blob in column number Col of row number Row in the current row set is fetched from the DataSource, irrespective of the value of the BlobDeferral property. In other words deferred blob fetching always applies if RowSetSize > 1.

THstmt.CellMemoryByName

property CellMemoryByName[ColName: **String**;
Row: SQLUSMALLINT]: TMemoryStream;

property TMemoryStream* CellMemoryByName[**AnsiString** ColName][SQLUSMALLINT Row];

Description

Performs the same function as the CellMemory property, except that it takes a column name instead of a column number to indicate a column in the result set.

CellMemoryByName[ColName] is equivalent to CellMemory[ColByName(ColName)].

THstmt.CellNull

property CellNull[Col, Row: SQLUSMALLINT]: **Boolean**;

property bool CellNull[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the ColNull property, except that it applies to row number Row in the current row set of the current result set.

THstmt.CellShortint

property CellShortint[Col, Row: SQLUSMALLINT]: **Shortint**;

property Shortint CellShortint[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the CellString property, except that it returns a shortint.

THstmt.CellShortintByName

property CellShortintByName[ColName: **String**;
Row: SQLUSMALLINT]: **Shortint**;

property Shortint CellShortintByName[**AnsiString** ColName][SQLUSMALLINT Row];

Description

Performs the same function as the CellShortint property, except that it takes a column name instead of a column number to indicate a column in the result set.

CellShortintByName[ColName] is equivalent to CellShortint[ColByName(ColName)].

THstmt.CellSingle

property CellSingle[Col, Row: SQLUSMALLINT]: **Single**;

property float CellSingle[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the CellString property, except that it returns a single.

THstmt.CellSingleByName

property CellSingleByName[ColName: **String**;
Row: SQLUSMALLINT]: **Single**;

property float CellSingleByName[**AnsiString** ColName][SQLUSMALLINT Row];

Description

Performs the same function as the CellSingle property, except that it takes a column name instead of a column number to indicate a column in the result set.

CellSingleByName[ColName] is equivalent to CellSingle[ColByName(ColName)].

THstmt.CellSize

property CellSize[Col, Row: SQLUSMALLINT]: SQLINTEGER;

property SQLINTEGER CellSize[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the ColSize property, except that it applies to row number Row in the current row set of the current result set.

THstmt.CellSmallint

property CellSmallint[Col, Row: SQLUSMALLINT]: **Smallint**;

property short CellSmallint[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the CellString property, except that it returns a smallint.

THstmt.CellSmallintByName

```
property CellSmallintByName[ColName: String;
                            Row: SQLUSMALLINT]: Smallint;
```

property short CellSmallintByName[**AnsiString** ColName][SQLUSMALLINT Row];

Description

Performs the same function as the CellSmallint property, except that it takes a column name instead of a column number to indicate a column in the result set.

CellSmallintByName[ColName] is equivalent to CellSmallint[ColByName(ColName)].

THstmt.CellStream

```
procedure CellStream(Col, Row: SQLUSMALLINT;
                     Stream: TStream);
```

```
void CellStream(SQLUSMALLINT Col,
                SQLUSMALLINT Row,
                TStream* Stream);
```

Description

Performs the same function as the ColStream method, except that it applies to row number Row in the current RowSet of the current result set.

THstmt.CellString

property CellString[Col, Row: SQLUSMALLINT]: **String**;

property AnsiString CellString[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the ColString property, except that it applies to row number Row in the current RowSet of the current result set.

THstmt.CellStringByName

```
property CellStringByName[ColName: String;
                          Row: SQLUSMALLINT]: String;
```

property AnsiString CellStringByName[**AnsiString** ColName][SQLUSMALLINT Row];

Description

Performs the same function as the CellString property, except that it takes a column name instead of a column number to indicate a column in the result set.

CellStringByName[ColName] is equivalent to CellString[ColByName(ColName)].

THstmt.CellTime

property CellTime[Col, Row: SQLUSMALLINT]: TTime;

property TTime CellTime[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the CellString property, except that it returns an ODBC TTime.

THstmt.CellTimeByName

```
property CellTimeByName[ColName: String;
                        Row: SQLUSMALLINT]: TTime;
```

property TTime CellTimeByName[**AnsiString** ColName][SQLUSMALLINT Row];

Description

Performs the same function as the CellTime property, except that it takes a column name instead of a column number to indicate a column in the result set.

CellTimeByName[ColName] is equivalent to CellTime[ColByName(ColName)].

THstmt.CellTimeStamp

property CellTimeStamp[Col, Row: SQLUSMALLINT]: TTimeStamp;

property TTimeStamp CellTimeStamp[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the CellString property, except that it returns an ODBC TTimeStamp.

THstmt.CellTimeStampByName

property CellTimeStampByName[ColName: **String**;
Row: SQLUSMALLINT]: TTimeStamp;

property TTimeStamp CellTimeStampByName[**AnsiString** ColName][SQLUSMALLINT Row];

Description

Performs the same function as the CellTimeStamp property, except that it takes a column name instead of a column number to indicate a column in the result set.

CellTimeStampByName[ColName] is equivalent to CellTimeStamp[ColByName(ColName)].

THstmt.CellValue

property CellValue[Col, Row: SQLUSMALLINT]: SQLPOINTER;

property SQLPOINTER CellValue[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the ColValue method, except that it applies to row number Row in the current row set of the current result set.

THstmt.CellVariant

property CellVariant[Col, Row: SQLUSMALLINT]: **Variant**;

property Variant CellVariant[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the CellString property, except that it returns a variant.

THstmt.CellVariantByName

property CellVariantByName[ColName: **String**;
Row: SQLUSMALLINT]: **Variant**;

property Variant CellVariantByName[**AnsiString** ColName][SQLUSMALLINT Row];

Description

Performs the same function as the CellVariant property, except that it takes a column name instead of a column number to indicate a column in the result set.

CellVariantByName[ColName] is equivalent to CellVariant[ColByName(ColName)].

THstmt.CellWord

property CellWord[Col, Row: SQLUSMALLINT]: **Word**;

property Word CellWord[SQLUSMALLINT Col][SQLUSMALLINT Row];

Description

Performs the same function as the CellString property, except that it returns a word.

THstmt.CellWordByName

property CellWordByName[ColName: **String**;
Row: SQLUSMALLINT]: **Word**;

property Word CellWordByName[**AnsiString** ColName][SQLUSMALLINT Row];

Description

Performs the same function as the CellWord property, except that it takes a column name instead of a column number to indicate a column in the result set.

CellWordByName[ColName] is equivalent to CellWord[ColByName(ColName)].

THstmt.Close

procedure Close; **virtual**;

virtual void Close(**void**);

Description

Forces the result set to be closed and the result set cursors to be freed. This method is automatically called when a new SQL statement is prepared.

THstmt.CloseCursor

procedure CloseCursor; **virtual**;

virtual void CloseCursor(**void**);

Description

Performs the same functionality as the Close method, except that it does not free the bound parameters and columns as well.

THstmt.ColBoolean

property ColBoolean[Col: SQLUSMALLINT]: **Boolean**;

property bool ColBoolean[SQLUSMALLINT Col];

Description

Performs the same function as the ColString property, except that it returns the value of column number Col in the current row of the result set as a boolean. If Col is out of range, it returns False.

THstmt.ColBooleanByName

property ColBooleanByName[ColName: **String**]: **Boolean**;

property bool ColBooleanByName[**AnsiString** ColName];

Description

Performs the same function as the ColBoolean property, except that it takes a column name instead of a column number to indicate a column in the result set.

ColBooleanByName[ColName] is equivalent to ColBoolean[ColByName(ColName)].

THstmt.ColByName

function ColByName(ColName: **String**): SQLUSMALLINT;

SQLUSMALLINT ColByName(**AnsiString** ColName);

Description

Returns the column number of the given column name ColName.

Note:

If you have duplicate column names in your SQL statement, even if they are qualified with owner/table names, you must rename the duplicate columns in your SQL statement using the "AS" SQL operator to ensure that all the column names are unique, otherwise this method will only return the column number of the first column in the SQL statement that matches the given column name ColName.

Example:

Delphi Code

```
with Hstmt1 do
begin
  //select two "Name" columns from two tables and rename the second one to "StudentName"
  SQL:= 'SELECT Lecturers.Name, Students.Name AS StudentName FROM Lecturers,
Students';
  Prepare;
  Execute;

  //fetch the two columns by name
  while FetchNext do
    ListBox1.Items.Add(ColString[ColByName('Name')]+' lectures
'+ColString[ColByName('StudentName')]);
    //or
    ListBox1.Items.Add(ColStringByName['Name']+' lectures '+ColStringByName['StudentName']);
end;
```

C++ Code

```
//select two "Name" columns from two tables and rename the second one to "StudentName"
Hstmt1->SQL = "SELECT Lecturers.Name, Students.Name AS StudentName FROM Lecturers,
Students";
Hstmt1->Prepare();
Hstmt1->Execute();

//fetch the two columns by name
while (Hstmt1->FetchNext()) {
  ListBox1->Items->Add(Hstmt1->ColString[Hstmt1->ColByName("Name")]+" lectures "+Hstmt1-
>ColString[Hstmt1->ColByName("StudentName")]);
  //or
  ListBox1->Items->Add(Hstmt1->ColStringByName["Name"]+" lectures "+Hstmt1-
>ColStringByName["StudentName"]);
}
```

THstmt.ColByte

property ColByte[Col: SQLUSMALLINT]: **Byte**;

property Byte ColByte[SQLUSMALLINT Col];

Description

Performs the same function as the ColString property, except that it returns the value of column number Col in the current row of the result set as a byte. If Col is out of range, it returns 0.

THstmt.ColByteByName

property ColByteByName[ColName: **String**]: **Byte**;

property Byte ColByteByName[**AnsiString** ColName];

Description

Performs the same function as the ColByte property, except that it takes a column name instead of a column number to indicate a column in the result set.

ColByteByName[ColName] is equivalent to ColByte[ColByName(ColName)].

THstmt.ColCardinal

property ColCardinal[Col: SQLUSMALLINT]: **Cardinal**;

property Cardinal ColCardinal[SQLUSMALLINT Col];

Description

Performs the same function as the ColString property, except that it returns the value of column number Col in the current row of the result set as a cardinal. If Col is out of range, it returns 0.

THstmt.ColCardinalByName

property ColCardinalByName[ColName: **String**]: **Cardinal**;

property Cardinal ColCardinalByName[**AnsiString** ColName];

Description

Performs the same function as the ColCardinal property, except that it takes a column name instead of a column number to indicate a column in the result set.

ColCardinalByName[ColName] is equivalent to ColCardinal[ColByName(ColName)].

THstmt.ColCount

property ColCount: SQLSMALLINT;

property SQLSMALLINT ColCount;

Description

Returns the number of columns in the current result set of the component. If the result set is empty or no result set exists, it returns 0.

THstmt.ColDate

property ColDate[Col: SQLUSMALLINT]: TDate;

property TDate ColDate[SQLUSMALLINT Col];

Description

Performs the same function as the ColString property, except that it returns the value of column number Col in the current row of the result set as an ODBC TDate. If Col is out of range, it returns the date part of NullTS.

THstmt.ColDateByName

property ColDateByName[ColName: **String**]: TDate;

property TDate ColDateByName[**AnsiString** ColName];

Description

Performs the same function as the ColDate property, except that it takes a column name instead of a column number to indicate a column in the result set.

ColDateByName[ColName] is equivalent to ColDate[ColByName(ColName)].

THstmt.ColDouble

property ColDouble[Col: SQLUSMALLINT]: **Double**;

property double ColDouble[SQLUSMALLINT Col];

Description

Performs the same function as the ColString property, except that it returns the value of column number Col in the current row of the result set as a double. If Col is out of range, it returns 0.

THstmt.ColDoubleByName

property ColDoubleByName[ColName: **String**]: **Double**;

property double ColDoubleByName[**AnsiString** ColName];

Description

Performs the same function as the ColDouble property, except that it takes a column name instead of a column number to indicate a column in the result set.

ColDoubleByName[ColName] is equivalent to ColDouble[ColByName(ColName)].

THstmt.ColFormatMask

property ColFormatMask[Col: SQLUSMALLINT]: **String**;

property AnsiString ColFormatMask[SQLUSMALLINT Col];

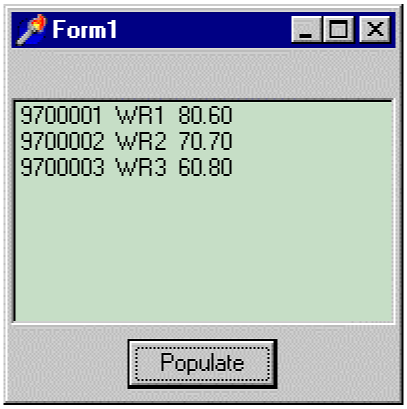
Description

Specifies the format mask to apply to column number Col, given the style as specified by the ColFormatStyle property. For each TFormatStyle value, the mask must be as follows:

fsNone	The format mask is ignored.
fsFloat	The format mask must be of the same format used together with the Delphi FormatFloat function.
fsDateTime	The format mask must be of the same format used together with the Delphi FormatDateTime function.
fsCustom	The format mask must be of the same format used together with the Delphi Format function.

Example:

Let's select the SNo, CCode and TMark fields from the Takes table into a listbox such that the TMark column is formatted as follows in the listbox:



```
9700001  WR1  80.60
9700002  WR2  70.70
9700003  WR3  60.80
```

Delphi Code

```
with Hstmt1 do
begin
  //set, prepare and execute SQL statement
  SQL:= 'SELECT SNo, CCode, TMark FROM Takes';
  Prepare;
  Execute;

  //set the format style and mask to format a float to 2 decimal places
  ColFormatStyle[ColByName('TMark')]:= fsFloat;
  ColFormatMask[ColByName('TMark')]:= '0.00';

  //add the rows to a listbox
  while FetchNext do
    ListBox1.Items.Add(ColStringByName['SNo']+'  '+
                                        ColStringByName['CCode']+'  '+
                                   ColStringByName['TMark']);
end;
```

C++ Code

```
//set, prepare and execute SQL statement
Hstmt1->SQL = "SELECT SNo, CCode, TMark FROM Takes";
Hstmt1->Prepare();
Hstmt1->Execute();

//set the format style and mask to format a float to 2 decimal places
Hstmt1->ColFormatStyle[Hstmt1->ColByName("TMark")] = fsFloat;
Hstmt1->ColFormatMask[Hstmt1->ColByName("TMark")] = "0.00";

//add the rows to a listbox
while (Hstmt1->FetchNext()) {
  ListBox1->Items->Add(Hstmt1->ColStringByName["SNo"]+"  "+
                                    Hstmt1->ColStringByName["CCode"]+"  "+
                                    Hstmt1->ColStringByName["TMark"]);
}
```

THstmt.ColFormatStyle

TFormatStyle = (fsNone, fsFloat, fsDateTime, fsCustom);

property ColFormatStyle[Col: SQLUSMALLINT]: TFormatStyle;

TFormatStyle {fsNone, fsFloat, fsDateTime, fsCustom};

property TFormatStyle ColFormatStyle[SQLUSMALLINT Col];

Description

Specifies the format style that must be applied to column number Col, given the mask as specified by the ColFormatMask property. It can be set to one of the following TFormatStyle values:

fsNone	Don't format the column.
fsFloat	Apply a real number format to the column.
fsDateTime	Apply a date and time format to the column.
fsCustom	Apply a custom format to the column.

THstmt.ColIgnore

property ColIgnore[Col: SQLUSMALLINT]: **Boolean**;

property bool ColIgnore[SQLUSMALLINT Col];

Description

This property allows you to set a column to be ignored when the current row is updated or deleted at the DataSource. It replaces the need to make use of the ColSize property of the Hstmt component for the same effect.

Example:

Delphi Code

```
var
  b: Boolean;
begin
  b:= Hstmt.ColIgnore[3];
  //is equivalent to
  b:= Hstmt.ColSize[3] = SQL_IGNORE;

  Hstmt.ColIgnore[3]:= True;
  //is equivalent to
  Hstmt.ColSize[3]:= SQL_IGNORE;
end;
```

C++ Code

```
bool b;

b = Hstmt->ColIgnore[3];
//is equivalent to
b = Hstmt->ColSize[3] == SQL_IGNORE;

Hstmt->ColIgnoreNull[3] = True;
//is equivalent to
Hstmt->ColSize[3] = SQL_IGNORE;
```

THstmt.ColInt64

property ColInt64[Col: SQLUSMALLINT]: **Int64**;

property __int64 ColInt64[SQLUSMALLINT Col];

Description

Performs the same function as the ColString property, except that it returns the value of column number Col in the current row of the result set as an int64. If Col is out of range, it returns 0.

THstmt.ColInt64ByName

property ColInt64ByName[ColName: **String**]: **Int64**;

property __int64 ColInt64ByName[**AnsiString** ColName];

Description

Performs the same function as the ColInt64 property, except that it takes a column name instead of a column number to indicate a column in the result set.

ColInt64ByName[ColName] is equivalent to ColInt64[ColByName(ColName)].

THstmt.ColInteger

property ColInteger[Col: SQLUSMALLINT]: **Integer**;

property int ColInteger[SQLUSMALLINT Col];

Description

Performs the same function as the ColString property, except that it returns the value of column number Col in the current row of the result set as an integer. If Col is out of range, it returns 0.

THstmt.ColIntegerByName

property ColIntegerByName[ColName: **String**]: **Integer**;

property int ColIntegerByName[**AnsiString** ColName];

Description

Performs the same function as the ColInteger property, except that it takes a column name instead of a column number to indicate a column in the result set.

ColIntegerByName[ColName] is equivalent to ColInteger[ColByName(ColName)].

THstmt.ColLongint

property ColLongint[Col: SQLUSMALLINT]: **Longint**;

property long ColLongint[SQLUSMALLINT Col];

Description

Performs the same function as the ColString property, except that it returns the value of column number Col in the current row of the result set as a longint. If Col is out of range, it returns 0.

THstmt.ColLongintByName

property ColLongintByName[ColName: **String**]: **Longint**;

property long ColLongintByName[**AnsiString** ColName];

Description

Performs the same function as the ColLongint property, except that it takes a column name instead of a column number to indicate a column in the result set.

ColLongintByName[ColName] is equivalent to ColLongint[ColByName(ColName)].

THstmt.ColLongword

property ColLongword[Col: SQLUSMALLINT]: **Longword**;

property Longword ColLongword[SQLUSMALLINT Col];

Description

Performs the same function as the ColString property, except that it returns the value of column number Col in the current row of the result set as a longword. If Col is out of range, it returns 0.

THstmt.ColLongwordByName

property ColLongwordByName[ColName: **String**]: **Longword**;

property Longword ColLongwordByName[**AnsiString** ColName];

Description

Performs the same function as the ColLongword property, except that it takes a column name instead of a column number to indicate a column in the result set.

ColLongwordByName[ColName] is equivalent to ColLongword[ColByName(ColName)].

THstmt.ColMemory

property ColMemory[Col: SQLUSMALLINT]: TMemoryStream;

property TMemoryStream* ColMemory[SQLUSMALLINT Col];

Description

Performs the same function as the ColString property, except that it returns the value of column number Col in the current row of the result set as a TMemoryStream. If Col is out of range, or no memory stream is allocated for the column, it returns nil. Memory streams are usually allocated for blob type columns. The memory stream is freed by the statement component, so it shouldn't be freed manually.

THstmt.ColMemoryByName

property ColMemoryByName[ColName: **String**]: TMemoryStream;

property TMemoryStream* ColMemoryByName[**AnsiString** ColName];

Description

Performs the same function as the ColMemory property, except that it takes a column name instead of a column number to indicate a column in the result set.

ColMemoryByName[ColName] is equivalent to ColMemory[ColByName(ColName)].

THstmt.ColNames

property ColNames: TStringList;

property TStringList* ColNames;

Description

Returns a list of the column names of the currently executed SQL statement. The list is zero-based, unlike the ColValue methods which are one-based.

THstmt.ColNull

property ColNull[Col: SQLUSMALLINT]: **Boolean**;

property bool ColNull[SQLUSMALLINT Col];

Description

This property allows you to check if a column is null or not, and set a column to be updated or inserted as null at a DataSource. It replaces the need to make use of the ColSize property of the Hstmt component for the same effect.

Example:

Delphi Code

```
var
  b: Boolean;
begin
  b:= Hstmt.ColNull[3];
  //is equivalent to
  b:= Hstmt.ColSize[3] = SQL_NULL_DATA;

  Hstmt.ColNull[3]:= True;
  //is equivalent to
  Hstmt.ColSize[3]:= SQL_NULL_DATA;
end;
```

C++ Code

```
bool b;

b = Hstmt->ColNull[3];
//is equivalent to
b = Hstmt->ColSize[3] == SQL_NULL_DATA;

Hstmt->ColNull[3] = True;
//is equivalent to
Hstmt->ColSize[3] = SQL_NULL_DATA;
```

THstmt.ColNullable

property ColNullable[Col: SQLUSMALLINT]: SQLSMALLINT;

property SQLSMALLINT ColNullable[SQLUSMALLINT Col];

Description

Returns a value which indicates whether the column allows NULL values. It can be one of the following values:

SQL_NO_NULLS	The column does not allow NULL values.
SQL_NULLABLE	The column allows NULL values.
SQL_NULLABLE_UNKNOWN	The driver cannot determine if the column allows NULL values.

THstmt.ColPrecision

property ColPrecision[Col: SQLUSMALLINT]: SQLUINTEGER;

property SQLUINTEGER ColPrecision[SQLUSMALLINT Col];

Description

Returns the size of the column at the data source. If the size cannot be determined, it returns 0.

THstmt.ColPrimary

property ColPrimary[Col: SQLUSMALLINT]: **Boolean**;

property bool ColPrimary[SQLUSMALLINT Col];

Description

Used to indicate which columns in the result set are the primary key columns. The primary columns are used in Level 3 SQL Generation to generate an update or delete statement with a primary key. This property can be set anytime after the select SQL statement was prepared. Set to True to indicate a primary column. The default value is False. Generally it is not necessary to set this property because:

1. Your ODBC Driver might support the level 2 ODBC functionality needed to perform positional operations (Level 1 and 2 SQL Generation).
2. Your ODBC Driver might be able to retrieve the primary columns of the table to generate an insert, update or delete statement (Level 3 SQL Generation). This will of course only work if the target table in the database is set up with a primary key.

If this property is set manually, it overrides automatic detection of the primary columns.

Example:

Delphi Code

```
with Hstmt1 do
begin
```

```
  //set, prepare and execute a select SQL statement
  SQL:= 'SELECT SNo, CCode, TMark FROM Takes';
  Prepare;
  Execute;

  //indicate the primary key to use in Level 3 SQL Generation
  ColPrimary[ColByName('SNo')]:= True;
  ColPrimary[ColByName('CCode')]:= True;
end;
```

C++ Code

```
//set, prepare and execute a select SQL statement
Hstmt1->SQL = "SELECT SNo, CCode, TMark FROM Takes";
Hstmt1->Prepare();
Hstmt1->Execute();

//indicate the primary key to use in Level 3 SQL Generation
Hstmt1->ColPrimary[Hstmt1->ColByName("SNo")] = True;
Hstmt1->ColPrimary[Hstmt1->ColByName("CCode")] = True;
```

THstmt.ColScale

property ColScale[Col: SQLUSMALLINT]: SQLSMALLINT;

property SQLSMALLINT ColScale[SQLUSMALLINT Col];

Description

Returns the number of decimal digits of the column at the data source. If the number of decimal digits cannot be determined or is not applicable, it returns 0.

THstmt.ColShortint

property ColShortint[Col: SQLUSMALLINT]: **Shortint**;

property Shortint ColShortint[SQLUSMALLINT Col];

Description

Performs the same function as the ColString property, except that it returns the value of column number Col in the current row of the result set as a shortint. If Col is out of range, it returns 0.

THstmt.ColShortintByName

property ColShortintByName[ColName: **String**]: **Shortint**;

property Shortint ColShortintByName[**AnsiString** ColName];

Description

Performs the same function as the ColShortint property, except that it takes a column name instead of a column number to indicate a column in the result set.

ColShortintByName[ColName] is equivalent to ColShortint[ColByName(ColName)].

THstmt.ColSingle

property ColSingle[Col: SQLUSMALLINT]: **Single**;

property float ColSingle[SQLUSMALLINT Col];

Description

Performs the same function as the ColString property, except that it returns the value of column number Col in the current row of the result set as a single. If Col is out of range, it returns 0.

THstmt.ColSingleByName

property ColSingleByName[ColName: **String**]: **Single**;

property float ColSingleByName[**AnsiString** ColName];

Description

Performs the same function as the ColSingle property, except that it takes a column name instead of a column number to indicate a column in the result set.

ColSingleByName[ColName] is equivalent to ColSingle[ColByName(ColName)].

THstmt.ColSize

property ColSize[Col: SQLUSMALLINT]: SQLINTEGER;

property SQLINTEGER ColSize[SQLUSMALLINT Col];

Description

Used to retrieve the size of column number Col in the current row of the result set, or to set the new size of a new value for column number Col, with the goal to do a positional insert or update of the row. If Col is out of range, it returns 0.

If the size is set to the constant SQL_IGNORE, the column will not be updated when a positional update is done. This is to prevent the component from trying to update columns at the DataSource which are not updatable, such as primary key columns, otherwise the DataSource will prevent the component from updating that column, which will result in the whole row being rejected by the DataSource. This is equivalent to using the ColIgnore property.

If the size of a column is equal to the constant SQL_NULL_DATA, the value of the column in the current row is NULL. The size of the column can also be set to SQL_NULL_DATA to insert a NULL value for the column when the row is inserted or updated. This is equivalent to the ColNull property.

THstmt.ColSmallint

property ColSmallint[Col: SQLUSMALLINT]: **Smallint**;

property short ColSmallint[SQLUSMALLINT Col];

Description

Performs the same function as the ColString property, except that it returns the value of column number Col in the current row of the result set as a smallint. If Col is out of range, it returns 0.

THstmt.ColSmallintByName

property ColSmallintByName[ColName: **String**]: **Smallint**;

property short ColSmallintByName[**AnsiString** ColName];

Description

Performs the same function as the ColSmallint property, except that it takes a column name instead of a column number to indicate a column in the result set.

ColSmallintByName[ColName] is equivalent to ColSmallint[ColByName(ColName)].

THstmt.ColStream

```
procedure ColStream(Col: SQLUSMALLINT;
                    Stream: TStream);
```

```
void ColStream(SQLUSMALLINT Col,
               TStream* Stream);
```

Description

Used to retrieve blob fields from the database directly to a file stream or any other TStream descendant, without caching them in memory as well. This method is best used together with blob deferral. If Col is out of range, the stream is left unchanged.

THstmt.ColString

property ColString[Col: SQLUSMALLINT]: **String**;

property AnsiString ColString[SQLUSMALLINT Col];

Description

Returns the value of column number Col in the current row of the result set as a string, or is used to set the value of the column with the goal to do a positional insert or update of the row. If Col is out of range, it returns an empty string.

THstmt.ColStringByName

property ColStringByName[ColName: **String**]: **String**;

property AnsiString ColStringByName[**AnsiString** ColName];

Description

Performs the same function as the ColString property, except that it takes a column name instead of a column number to indicate a column in the result set.

ColStringByName[ColName] is equivalent to ColString[ColByName(ColName)].

THstmt.ColTime

property ColTime[Col: SQLUSMALLINT]: TTime;

property TTime ColTime[SQLUSMALLINT Col];

Description

Performs the same function as the ColString property, except that it returns the value of column number Col in the current row of the result set as an ODBC TTime. If Col is out of range, it returns the time part of NullTS.

THstmt.ColTimeByName

property ColTimeByName[ColName: **String**]: TTime;

property TTime ColTimeByName[**AnsiString** ColName];

Description

Performs the same function as the ColTime property, except that it takes a column name instead of a column number to indicate a column in the result set.

ColTimeByName[ColName] is equivalent to ColTime[ColByName(ColName)].

THstmt.ColTimeStamp

property ColTimeStamp[Col: SQLUSMALLINT]: TTimeStamp;

property TTimeStamp ColTimeStamp[SQLUSMALLINT Col];

Description

Performs the same function as the ColString property, except that it returns the value of column number Col in the current row of the result set as an ODBC TTimeStamp. If Col is out of range, it returns NullTS.

THstmt.ColTimeStampByName

property ColTimeStampByName[ColName: **String**]: TTimeStamp;

property TTimeStamp ColTimeStampByName[**AnsiString** ColName];

Description

Performs the same function as the ColTimeStamp property, except that it takes a column name instead of a column number to indicate a column in the result set.

ColTimeStampByName[ColName] is equivalent to ColTimeStamp[ColByName(ColName)].

THstmt.ColType

property ColType(Col: SQLUSMALLINT): SQLSMALLINT;

property SQLSMALLINT ColType[SQLUSMALLINT Col];

Description

Returns the storage data type (data type of column as in memory) of column number Col in the current row of the result set. If Col is out of range, it returns 0.

THstmt.ColValue

property ColValue[Col: SQLUSMALLINT]: SQLPOINTER;

property SQLPOINTER ColValue[SQLUSMALLINT Col];

Description

Performs the same function as the ColString property, except that it returns a pointer to the value of column number Col in the current row of the result set. If Col is out of range, it returns nil. Modifying the memory pointed to by the returned pointer has a similar effect as assigning a new value to the ColString property.

THstmt.ColVariant

property ColVariant[Col: SQLUSMALLINT]: **Variant**;

property Variant ColVariant[SQLUSMALLINT Col];

Description

Performs the same function as the ColString property, except that it returns the value of column number Col in the current row of the result set as a variant. If the column is null, it returns the variant value Unassigned.

THstmt.ColVariantByName

property ColVariantByName[ColName: **String**]: **Variant**;

property Variant ColVariantByName[**AnsiString** ColName];

Description

Performs the same function as the ColVariant property, except that it takes a column name instead of a column number to indicate a column in the result set.

ColVariantByName[ColName] is equivalent to ColVariant[ColByName(ColName)].

THstmt.ColWord

property ColWord[Col: SQLUSMALLINT]: **Word**;

property Word ColWord[SQLUSMALLINT Col];

Description

Performs the same function as the ColString property, except that it returns the value of column number Col in the current row of the result set as a word. If Col is out of range, it returns 0.

THstmt.ColWordByName

property ColWordByName[ColName: **String**]: **Word**;

property Word ColWordByName[**AnsiString** ColName];

Description

Performs the same function as the ColWord property, except that it takes a column name instead of a column number to indicate a column in the result set.

ColWordByName[ColName] is equivalent to ColWord[ColByName(ColName)].

THstmt.ConcurrencyType

property ConcurrencyType: SQLUINTEGER;

property SQLUINTEGER ConcurrencyType;

Description

Used to set the locking strategy used by the result set cursor at the DataSource. The default value is SQL_CONCUR_READ_ONLY. The property can be set to one of the following values:

SQL_CONCUR_READ_ONLY	Cursor is read-only. No updates are allowed.
SQL_CONCUR_LOCK	Cursor uses the lowest level of locking sufficient to ensure that the row can be updated.

SQL_CONCUR_ROWVER	Cursor uses optimistic concurrency control, comparing row versions.
SQL_CONCUR_VALUES	Cursor uses optimistic concurrency control, comparing values.

THstmt.CursorType

property CursorType: SQLUINTEGER;

property SQLUINTEGER CursorType;

Description

Used to set the way a cursor can scroll through the result set. The default value is SQL_CURSOR_FORWARD_ONLY. The property can be set to one of the following values:

SQL_CURSOR_FORWARD_ONLY	The cursor only scrolls forward.
SQL_CURSOR_STATIC	The data and rows which makes up the result set is static.
SQL_CURSOR_KEYSET_DRIVEN	The data in the result set is dynamic, but the rows which makes up the result set is static.
SQL_CURSOR_DYNAMIC	The data and rows which makes up the result set is dynamic.

THstmt.DoAfterExecute

procedure DoAfterExecute; **virtual**;

virtual void DoAfterExecute(**void**);

Description

Calls the OnAfterExecute event.

THstmt.DoAfterFetch

procedure DoAfterFetch; **virtual**;

virtual void DoAfterFetch(**void**);

Description

Calls the OnAfterFetch event.

THstmt.DoAfterPrepare

procedure DoAfterPrepare; **virtual**;

virtual void DoAfterPrepare(**void**);

Description

Calls the OnAfterPrepare event.

THstmt.DoBeforeExecute

procedure DoBeforeExecute; **virtual**;

virtual void DoBeforeExecute(**void**);

Description

Calls the OnBeforeExecute event.

THstmt.DoBeforeFetch

procedure DoBeforeFetch; **virtual**;

virtual void DoBeforeFetch(**void**);

Description

Calls the OnBeforeFetch event.

THstmt.DoBeforePrepare

procedure DoBeforePrepare; **virtual**;

virtual void DoBeforePrepare(**void**);

Description

Calls the OnBeforePrepare event.

THstmt.DoDelete

procedure DoDelete; **virtual**;

virtual void DoDelete(**void**);

Description

Used to do a positional delete in the result set. Instead of specifying a delete statement to delete a row at the DataSource, a result set can be selected from the DataSource and then a positional delete can be made. This method also calls the OnDelete event.

The DoDelete method will make use of one of the 4 Levels of SQL Generation to perform the delete.

THstmt.DoInsert

procedure DoInsert; **virtual**;

virtual void DoInsert(**void**);

Description

Used to do a positional insert in the result set. New values are assigned to the columns of a row in the result set, and then DoInsert is called to insert the row at the DataSource. Instead of specifying an insert statement to insert a row at the DataSource, a result set can be selected from the DataSource and then a positional insert can be made. This method also calls the OnInsert event.

The DoInsert method will make use of one of the 4 Levels of SQL Generation to perform the insert.

THstmt.DoRefresh

procedure DoRefresh; **virtual**;

virtual void DoRefresh(**void**);

Description

Used to do a positional refresh in the result set. This method also calls the OnRefresh event.

The DoRefresh method will make use of one of the 4 Levels of SQL Generation to perform the refresh.

THstmt.DoRowCount

function DoRowCount: Integer; **virtual**;

virtual int DoRowCount(**void**);

Description

Calls the OnRowCount event.

THstmt.DoUpdate

procedure DoUpdate; **virtual**;

virtual void DoUpdate(**void**);

Description

Used to do a positional update in the result set. New values are assigned to the columns of a row in the result set, and then DoUpdate is called to update the row at the DataSource. Instead of specifying an update statement to update a row at the DataSource, a result set can be selected from the DataSource and then a positional update can be made. This method also calls the OnUpdate event.

The DoUpdate method will make use of one of the 4 Levels of SQL Generation to perform the update.

THstmt.EmptyStringToNull

TEmptyToNull = (enNever, enAlways, enIfNullable);

property EmptyStringToNull: TEmptyToNull;

TEmptyToNull {enNever, enAlways, enIfNullable};

property TEmptyToNull EmptyStringToNull;

Description

Used to control the way empty strings are interpreted by ODBCExpress. The default value is enNever. The property can have one of the following values:

enNever: Empty strings are never converted to null values in the database, in other words a distinction is made between empty and null strings.
enAlways: Empty strings are always converted to null values in the database.
enIfNeeded: Empty strings are only converted to null values in the database if ODBCExpress detects that the columns in question are nullable.

Note:

This currently only works for string columns used in positional insert, update and delete operations, and not for string parameters.

THstmt.ExecAsync

property ExecAsync: **Boolean**;

property bool ExecAsync;

Description

Allows you to make use of cancelable queries. The default value is False. Setting this property to True will enable asynchronous execution. The Execute method, though still a blocking call, will then process application messages to allow you to call the AbortQuery method from the same (or different) thread to cancel query processing. Calling the AbortQuery method will cause the Execute method to return by raising the silent EABORT exception. If the AbortQuery method is not called the Execute method will return normally when query processing has completed. In both cases the Executed property will be set to True after the Execute returns.

Setting this property to True also causes the application messages to be processed during execution, so this can also be used to perform other operations and user interaction while a SQL statement is executed.

Example:

Delphi Code

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  //enable asynchronous execution
  Hstmt1.ExecAsync:= True;

  //execute time-consuming query
  Hstmt1.SQL:= 'SELECT * FROM Students';
  Hstmt1.Execute;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  //cancel query processing
  Hstmt1.AbortQuery;
end;
```

C++ Code

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  //enable asynchronous execution
  Hstmt1->ExecAsync = true;

  //execute time-consuming query
  Hstmt1->SQL = "SELECT * FROM Students";
  Hstmt1->Execute();
}

void __fastcall TForm1::Button2Click(TObject *Sender)
{
  //cancel query processing
  Hstmt1->AbortQuery();
}
```

THstmt.Execute

procedure Execute;

void Execute(**void**);

Description

Used to execute the SQL statement at the DataSource. If this method is called without preparing the SQL statement, the SQL statement will directly be executed at the database without preparing it.

THstmt.Executed

property Executed: **Boolean**;

property bool Executed;

Description

Allows you to keep track of the executed state of a SQL statement. The default value is False. Setting this property to True is equivalent to calling the Execute method.

THstmt.FetchAbsolute

function FetchAbsolute(Row: SQLINTEGER): **Boolean**;

bool FetchAbsolute(SQLINTEGER Row);

Description

Fetches row number Row in the result set. This method only works if there is a static-type scrollable cursor on the result set. A positive value will fetch absolute from the beginning of the result set and a negative value will fetch absolute from the end of the result set.

THstmt.FetchBookmark

function FetchBookmark(Bookmark: SQLPOINTER): **Boolean**;

bool FetchBookmark(SQLPOINTER Bookmark);

Description

Fetches the row in the result set which corresponds to the given Bookmark, retrieved using the Bookmark property. This method only works if there is a scrollable cursor on the result set.

THstmt.FetchFirst

function FetchFirst: **Boolean**;

bool FetchFirst(**void**);

Description

Fetches the first row in the result set. This method only works if there is a scrollable cursor on the result set.

THstmt.FetchLast

function FetchLast: **Boolean**;

bool FetchLast(**void**);

Description

Fetches the last row in the result set. This method only works if there is a scrollable cursor on the result set.

THstmt.FetchNext

function FetchNext: **Boolean**;

bool FetchNext(**void**);

Description

Fetches the next row in the result set. This method works with any cursor on the result set. It fetches the first row if the result set cursor is positioned before the first row.

THstmt.FetchPrev

function FetchPrev: **Boolean**;

bool FetchPrev(**void**);

Description

Fetches the previous row in the result set. This method only works if there is a scrollable cursor on the result set. It fetches the last row if the result set cursor is positioned after the last row.

THstmt.FetchRelative

function FetchRelative(Row: SQLINTEGER): **Boolean**;

bool FetchRelative(SQLINTEGER Row);

Description

Fetches the row at offset Row to the current Row in the result set. This method only works if there is a static-type scrollable cursor on the result set. A positive value will fetch forward from the current row in the result set and a negative value will fetch backward from the current row in the result set.

THstmt.FreeBookmark

procedure FreeBookmark(ABookmark: SQLPOINTER);

void FreeBookmark(SQLPOINTER ABookmark);

Description

Used to free the bookmark values returned by the Bookmark property after they are not needed anymore.

THstmt.Handle

property Handle: SQLHSTMT;

property SQLHSTMT Handle;

Description

This returns the statement handle contained in the component. This handle can be used to call other ODBC functions which are not directly supported by the statement component.

THstmt.hDbc

property hDbc: THdbc;

property THdbc* hDbc;

Description

Used to set the parent connection component of the statement component. The default value is nil. When a statement component is thrown into a data module or onto a form, it will take on the value of the first connection component thrown into the data module or onto the form.

THstmt.IgnoreCols

procedure IgnoreCols(Cols: **array of String**);

void IgnoreCols(**AnsiString*** Cols,
const int Cols_Size);

Description

Indicates the columns in a result set to be ignored during positional Inserts and Updates. This method makes multiple calls to the ColIgnore[Col] property.

Example:

Delphi Code

```
Hstmt1.IgnoreCols(['SNo', 'CCode']);

//is equivalent to the two calls

Hstmt1.ColIgnore[Hstmt1.ColByName('SNo')]:= True;
Hstmt1.ColIgnore[Hstmt1.ColByName('CCode')]:= True;
```

C++ Code

```
AnsiString IgnoreCols[] = {"SNo", "CCode"};
Hstmt1->IgnoreCols(IgnoreCols, ARRAYSIZE(IgnoreCols)-1);

//is equivalent to the two calls
```

```
Hstmt1->ColIgnore[Hstmt1->ColByName("SNo")] = true;
Hstmt1->ColIgnore[Hstmt1->ColByName("CCode")] = true;
```

THstmt.MaxRows

property MaxRows: SQLUINTEGER;

property SQLUINTEGER MaxRows;

Description

Allows you to limit the number of rows returned in a result set. The default value is 0, which indicates that the number of rows in a result set is not limited. If set to a value > 0, and the number of rows returned by a result set is equal to the value of this property, it means that there might be more rows available to the result set if it was not limited.

Whether this functionality is supported is dependant on the ODBC driver used.

THstmt.MoreResults

function MoreResults: **Boolean**;

bool MoreResults(**void**);

Description

This method is used to determine whether more result sets are available on the Hstmt component and if so it will initialize processing for the next result set. An Hstmt component typically returns multiple result sets when:

1. a stored procedure executed by the Hstmt component returns more than one result set or
2. when multiple SQL statements are specified for the SQL property if the Hstmt component.

Multiple SQL statements can be specified for the SQL property by separating each SQL statement with a semi-colon ";", provided that the ODBC Driver used supports execution of multiple SQL statements.

Each time the MoreResults function returns True, the Hstmt component will be ready to process the next result set. Since the Hstmt component is always ready to process the first result set after execution, the MoreResults function cannot be used to initialize the first result set.

Example:

This example illustrates the logic of the MoreResults function.

Delphi Code

```
with Hstmt1 do
begin
  SQL:= 'SELECT * FROM Students;  SELECT * FROM Courses';
  Prepare;
  Execute;

  repeat
```

```
    while FetchNext do
      //do something with the data
  until not MoreResults;
end;
```

C++ Code

```
Hstmt1->SQL = "SELECT * FROM Students;  SELECT * FROM Courses";
Hstmt1->Prepare();
Hstmt1->Execute();

do {
  while (Hstmt1->FetchNext()) {
    //do something with the data
  }
} while (Hstmt1->MoreResults());
```

THstmt.NoRowsAffected

type TNoRowsAffected = **set of** (nrByInsert, nrByUpdate, nrByDelete, nrByRefresh);

property NoRowsAffected: TNoRowsAffected;

enum temp {nrByInsert, nrByUpdate, nrByDelete, nrByRefresh};
typedef Set<temp, nrByInsert, nrByRefresh> TNoRowsAffected;

property TNoRowsAffected NoRowsAffected;

Description

Allows you to control whether exceptions must be raised when no rows are affected by a positional insert, update or delete operation (level 1 to level 3 SQL Generation). It can be one or more of the following values:

nrByInsert	Raise an exception when no rows are affected by a positional insert operation. This should not be necessary to set since exceptions are normally raised when an insert fails.
nrByUpdate	Raise an exception when no rows are affected by a positional update operation.
nrByDelete	Raise an exception when no rows are affected by a positional delete operation. Normally you won't be interested in this scenario, since the row is already deleted.
nrByRefresh	Raise an exception when no rows are affected by a refresh operation. Not being able to refresh a row is normally not a serious problem.

By default the set only contains the value nrByUpdate.

THstmt.NullCols

procedure NullCols(Cols: **array of String**);

void NullCols(**AnsiString*** Cols,
const int Cols_Size);

Description

Sets a number of columns in a result set to null. This method makes multiple calls to the ColNull[Col] property.

Example:

Delphi Code

```
Hstmt1.NullCols(['SNo', 'CCode']);
```

```
//is equivalent to the two calls
```

```
Hstmt1.ColNull[Hstmt1.ColByName('SNo')]:= True;
Hstmt1.ColNull[Hstmt1.ColByName('CCode')]:= True;
```

C++ Code

```
AnsiString NullCols[] = {"SNo", "CCode"};
Hstmt1->NullCols(NullCols, ARRAYSIZE(NullCols)-1);
```

```
//is equivalent to the two calls
```

```
Hstmt1->ColNull[Hstmt1->ColByName("SNo")] = true;
Hstmt1->ColNull[Hstmt1->ColByName("CCode")] = true;
```

THstmt.OnDelete

```
TConfirmEvent = function (Sender: TObject;
                          DefaultMsg: String): Boolean of object;
```

```
property OnDelete: TConfirmEvent;
```

```
bool (closure* TConfirmEvent)(TObject* Sender,
                              AnsiString DefaultMsg);
```

```
property TConfirmEvent OnDelete;
```

Description

This event is called on each DoDelete made with the statement component. This event can be used to confirm the positional delete, for which a default message DefaultMsg is provided. If True is returned from the event, the positional delete will be performed, otherwise it will be skipped. This event can also be used to perform Level 4 SQL Generation, in which case False must be returned from the event to prevent the DoDelete method from performing one of the first three Levels of SQL Generation.

THstmt.OnInsert

```
TConfirmEvent = function (Sender: TObject;
                         DefaultMsg: String): Boolean of object;
```

```
property OnInsert: TConfirmEvent;
```

```
bool (closure* TConfirmEvent)(TObject* Sender,
```

AnsiString DefaultMsg);

property TConfirmEvent OnInsert;

Description

This event is called on each DoInsert made with the statement component. This event can be used to confirm the positional insert, for which a default message DefaultMsg is provided. If True is returned from the event, the positional insert will be performed, otherwise it will be skipped. This event can also be used to perform Level 4 SQL Generation, in which case False must be returned from the event to prevent the DoInsert method from performing one of the first three Levels of SQL Generation.

THstmt.OnRefresh

TConfirmEvent = **function** (Sender: TObject;
DefaultMsg: **String**): **Boolean of object**;

property OnRefresh: TConfirmEvent;

bool (**closure*** TConfirmEvent)(TObject* Sender,
AnsiString DefaultMsg);

property TConfirmEvent OnRefresh;

Description

This event is called on each DoRefresh made with the statement component. This event can be used to confirm the positional refresh, for which a default message DefaultMsg is provided. If True is returned from the event, the positional refresh will be performed, otherwise it will be skipped. This event can also be used to perform Level 4 SQL Generation, in which case False must be returned from the event to prevent the DoRefresh method from performing Level 1 of SQL Generation.

THstmt.OnRowCount

TRowCountEvent = **function** (Sender: TObject): **Integer** of object;

property OnRowCount: TRowCountEvent;

int (**closure*** TRowCountEvent)(TObject* Sender);

property TRowCountEvent OnRowCount;

Description

Allows the user to determine the row count when the RowCountMethod property is set to rcCustom. If the event returns a value < 0, or the event is not implemented, then one of the other row count methods are used.

THstmt.OnStatement

```
TStatementEvent = procedure (Sender: TObject;
                             Operation: Word;
                             var SQL: String) of object;
```

```
property OnStatement: TStatementEvent;
```

```
void (closure* TStatementEvent)(TObject* Sender,
                                Word Operation,
                                AnsiString &SQL);
```

```
property TStatementEvent OnStatement;
```

Description

This event can be used to inspect and possibly modify the SQL statements generated during positional Inserts, Updates and Deletes, if any. Any modifications of the SQL statement can be done, as long as the number of parameters and their order in the SQL statement are not changed.

THstmt.OnUpdate

```
TConfirmEvent = function (Sender: TObject;
                          DefaultMsg: String): Boolean of object;
```

```
property OnUpdate: TConfirmEvent;
```

```
bool (closure* TConfirmEvent)(TObject* Sender,
                              AnsiString DefaultMsg);
```

```
property TConfirmEvent OnUpdate;
```

Description

This event is called on each DoUpdate made with the statement component. This event can be used to confirm the positional update, for which a default message DefaultMsg is provided. If True is returned from the event, the positional update will be performed, otherwise it will be skipped. This event can also be used to perform Level 4 SQL Generation, in which case False must be returned from the event to prevent the DoUpdate method from performing one of the first three Levels of SQL Generation.

THstmt.ParamByName

```
function ParamByName(ParamName: String): SQLUSMALLINT;
```

```
SQLUSMALLINT ParamByName(AnsiString ParamName);
```

Description

Returns the parameter number of the given parameter name ParamName.

Note:

This method can only be used if all the parameters in the SQL statement are named parameters, in other words there cannot be any unnamed parameters in the SQL statement.

THstmt.ParamNames

property ParamNames: TStringList;

property TStringList* ParamNames;

Description

Returns a list of the parameter names of the currently prepared SQL statement. The list is zero-based, unlike the BindParam methods which are one-based.

THstmt.ParamNullable

property ParamNullable[Param: SQLUSMALLINT]: SQLSMALLINT;

property SQLSMALLINT ParamNullable[SQLUSMALLINT Param];

Description

Returns a value which indicates whether the parameter allows NULL values. It can be one of the following values:

SQL_NO_NULLS	The parameter does not allow NULL values.
SQL_NULLABLE	The parameter allows NULL values.
SQL_NULLABLE_UNKNOWN	The driver cannot determine if the parameter allows NULL values.

THstmt.ParamPrecision

property ParamPrecision[Param: SQLUSMALLINT]: SQLUINTEGER;

property SQLUINTEGER ParamPrecision[SQLUSMALLINT Param];

Description

Returns the size of the column or expression of the corresponding parameter as defined by the data source. If the size cannot be determined, it returns 0.

THstmt.ParamScale

property ParamScale[Param: SQLUSMALLINT]: SQLSMALLINT;

property SQLSMALLINT ParamScale[SQLUSMALLINT Param];

Description

Returns the number of decimal digits of the column or expression of the corresponding parameter as defined by the data source. If the number of decimal digits cannot be determined or is not applicable, it returns 0.

THstmt.ParamSize

property ParamSize[Param: SQLUSMALLINT]: SQLINTEGER;

property SQLINTEGER ParamSize[SQLUSMALLINT Param];

Description

Used to read and set the exact size of an input blob parameter number Param. It must be set after the blob's storage is bound using the BindParam or BindParamCore method. This property is automatically set when using the BindMemory, BindText and BindBinary methods.

Example:

Delphi Code

```
var
  M: TMemoryStream;
begin
  M:= TMemoryStream.Create;

  //load picture into memory stream
  M.LoadFromFile('Student.PIC');

  with Hstmt1 do
  begin
    //prepare statement
    SQL:= 'INSERT INTO Students (SNo, SPic) VALUES ("9700001", ?)';
    Prepare;

    //bind memory stream and set parameter size
    BindParam(1, SQL_C_BINARY, M.Memory);
    ParamSize[1]:= M.Size;

    //or

    //bind memory stream and set parameter size in one step
    BindBinary(1, M);

    //insert blob into database
    Execute;
  end;

  M.Free;
end;
```

C++ Code

```
TMemoryStream* M ;

M = new TMemoryStream();
```

```
//load picture into memory stream
M->LoadFromFile("Student.PIC");

//prepare statement
Hstmt1->SQL = "INSERT INTO Students (SNo, SPic) VALUES ("9700001", ?)";
Hstmt1->Prepare();

//bind memory stream and set parameter size
Hstmt1->BindParam(1, SQL_C_BINARY, M->Memory);
Hstmt1->ParamSize[1] = M->Size;

//or

//bind memory stream and set parameter size in one step
Hstmt1->BindBinary(1, M);

//insert blob into database
Hstmt1->Execute();

delete M;
```

THstmt.ParamType

property ParamType: SQLSMALLINT;

property SQLSMALLINT ParamType;

Description

Used to specify the type of stored procedure parameter that is to be bound next. This property can be set to any of the following values, each indicating a stored procedure parameter type:

SQL_PARAM_INPUT	Indicates an IN parameter is to be bound next.
SQL_PARAM_OUTPUT	Indicates an OUT parameter is to be bound next.
SQL_PARAM_INPUT_OUTPUT	Indicates an IN/OUT parameter is to be bound next.

This property must be set immediately before the parameter is bound, and after the bound, the property will automatically be reset to the default value of SQL_PARAM_INPUT. Therefore, when binding an input parameter type, there's no need to set this property since the default value of ParamType is always SQL_PARAM_INPUT.

Example:

Delphi Code

```
//bind OUTPUT parameter
Hstmt1.ParamType:= SQL_PARAM_OUTPUT;
Hstmt1.BindParam(1, ...);

//bind INPUT/OUTPUT parameter
Hstmt1.ParamType:= SQL_PARAM_INPUT_OUTPUT;
Hstmt1.BindParam(2, ...);

//bind INPUT parameter without setting the ParamType property
```

```
Hstmt1.BindParam(3, ...);
```

C++ Code

```
//bind OUTPUT parameter
Hstmt1->ParamType = SQL_PARAM_OUTPUT;
Hstmt1->BindParam(1, ...);

//bind INPUT/OUTPUT parameter
Hstmt1->ParamType = SQL_PARAM_INPUT_OUTPUT;
Hstmt1->BindParam(2, ...);

//bind INPUT parameter without setting the ParamType property
Hstmt1->BindParam(3, ...);
```

THstmt.Prepare

procedure Prepare;

void Prepare(**void**);

Description

Prepares the SQL statement for execution at the DataSource. This function is called to start a new query. To allow for full interaction of the statement class with external ODBC function calls, this method is not automatically called by subsequent component method calls.

THstmt.Prepared

property Prepared: **Boolean**;

property bool Prepared;

Description

Allows you to keep track of the prepared state of a SQL statement. The default value is False. Setting this property to True is equivalent to calling the Prepare method.

THstmt.PrimaryCols

procedure PrimaryCols(Cols: **array of String**);

void PrimaryCols(**AnsiString*** Cols,
const int Cols_Size);

Description

Indicates the columns in a result set which form part of the primary key. This method makes multiple calls to the ColPrimary[Col] property.

Example:

Delphi Code

```
Hstmt1.PrimaryCols(['SNo', 'CCode']);

//is equivalent to the two calls

Hstmt1.ColPrimary[Hstmt1.ColByName('SNo')]:= True;
Hstmt1.ColPrimary[Hstmt1.ColByName('CCode')]:= True;
```

C++ Code

```
AnsiString PrimaryCols[] = {"SNo", "CCode"};
Hstmt1->PrimaryCols(PrimaryCols, ARRAYSIZE(PrimaryCols)-1);

//is equivalent to the two calls

Hstmt1->ColPrimary[Hstmt1->ColByName("SNo")] = true;
Hstmt1->ColPrimary[Hstmt1->ColByName("CCode")] = true;
```

THstmt.QueryTimeOut

property QueryTimeOut: SQLUINTEGER;

property SQLUINTEGER QueryTimeOut;

Description

Used to set the number of seconds after which a query at a DataSource must timeout. A value of 0 indicates that no timeout must occur. The default value is ODBC Driver dependent.

Example:

Delphi Code

```
//set timeout to occur after 20 seconds
Hstmt1.QueryTimeOut:= 20;
Hstmt1.Execute;
```

C++ Code

```
//set timeout to occur after 20 seconds
Hstmt1->QueryTimeOut = 20;
Hstmt1->Execute();
```

THstmt.RowCount

property RowCount: SQLINTEGER;

property SQLINTEGER RowCount;

Description

This property returns a very fast and accurate estimate of the number of rows that is in your result set. Instead of performing a time-consuming retrieval of the rows in your result it, the property either makes an ODBC call or performs a query at the database to determine the number of rows, making use of the current query and parameters. While it's safe to say that this property will in almost all cases return the exact number of rows in your result set, there might be cases in multi-user environments that the number of rows affected by your query changes from the time that your result set is build to the time that the number of rows in the result set is determined, but this would only be a very slight deviation of no more than a couple of rows.

If it cannot determine the number of rows using the above methods, it will fetch and count the number of rows returned. This is also an estimate of the number of rows in the result set as described above, since it will make use of another very fast forward-only result set to determine the number of rows.

You can control the method used to determine the number of rows using the RowCountMethod property. The MaxRows property and the DISTINCT, HAVING and UNION clauses are taken into consideration when determining the RowCount. However limiting the rows in a result set via SQL commands will not be taken into consideration.

THstmt.RowCountMethod

TRowCountMethod = (rcFunction, rcSelect, rcTraverse, rcCustom);

property RowCountMethod: TRowCountMethod;

TRowCountMethod {rcFunction, rcSelect, rcTraverse, rcCustom};

property TRowCountMethod RowCountMethod;

Description

Used to control the way the row count is determined. The default value is rcFunction. The property can have one of the following values:

rcFunction Uses an ODBC function to determine the row count, or one of the following methods if not supported.
rcCustom Allows the user to determine the row count via the OnRowCount event. If the OnRowCount event returns a value < 0, or the OnRowCount event is not implemented, then one of the following methods are used.
rcSelect Performs a "SELECT Count(*)" approximation, or the following method if not supported.
rcTraverse Returns an approximation by creating a temporary result set and counting the rows in this result set by retrieving them.

If detected that a method is not supported (in the above order), the following method will automatically be attempted. This property is useful for skipping a supported method which does not return the correct row count.

THstmt.RowFlag

property RowFlag[Row: SQLUSMALLINT]: SQLUSMALLINT;

property SQLUSMALLINT RowFlag[SQLUSMALLINT Row];

Description

Used to indicate the operation to be performed on row number Row in the current row set when the DoInsert, DoUpdate, DoDelete and DoRefresh methods are called. The default value is rfNone. It can be set to one of the following values:

rfNone	Indicates that the row must be ignored when any of the methods is called.
rfInsert	Indicates that the row must be inserted when DoInsert is called.
rfUpdate	Indicates that the row must be updated when DoUpdate is called.
rfDelete	Indicates that the row must be deleted when DoDelete is called.
rfRefresh	Indicates that the row must be refreshed when DoRefresh is called.

Note:

When RowSetSize is 1, the RowFlag property will be automatically set to the appropriate value when one of the methods is called.

THstmt.RowsAffected

property RowsAffected: SQLINTEGER;

property SQLINTEGER RowsAffected;

Description

Returns the number of rows affected at a DataSource by an insert, update or delete operation.

THstmt.RowSetSize

property RowSetSize: SQLUINTEGER;

property SQLUINTEGER RowSetSize;

Description

Used to set the number of rows that must be retrieved from a result set during each fetch. The default value is 1. Use this property to speed up row-retrieval from a DataSource. Play off speed against memory used when setting this property. When the RowSetSize is greater than one, rows retrieved can be accessed using the CellValue properties.

THstmt.RowsFetched

property RowsFetched: SQLUINTEGER;

property SQLUINTEGER RowsFetched;

Description

Returns the number of rows in the current row set of the current result set of the component, and is always less than or equal to the RowSetSize property.

Note:

This property generally does not indicate the number of rows in the current result set.

THstmt.RowStatus

property RowStatus[Row: SQLUSMALLINT]: SQLUSMALLINT;

property SQLUSMALLINT RowStatus[SQLUSMALLINT Row];

Description

This read-only property is used to determine the status of row number Row in the current row set of the current result set. It can be one of the following values:

SQL_ROW_SUCCESS	The row was successfully fetched and has not changed since it was last fetched.
SQL_ROW_SUCCESS_WITH_INFO	The row was successfully fetched and has not changed since it was last fetched, however a warning was returned about the row.
SQL_ROW_ADDED	The row was inserted.
SQL_ROW_UPDATED	The row was successfully fetched and has been updated since it was last fetched.
SQL_ROW_DELETED	The row has been deleted since it was last fetched.
SQL_ROW_ERROR	An error occurred while fetching the row.
SQL_ROW_NOROW	The rowset overlapped the end of the result set and no row was returned that corresponded to this element of the row status array.

THstmt.RowValid

property RowValid[Row: SQLUSMALLINT]: **Boolean**;

property bool RowValid[SQLUSMALLINT Row];

Description

Returns True if the RowStatus value of the row indicates a valid row, otherwise False.

The value of the property is equivalent to the result of the following statement:

```
RowStatus[Row] in [SQL_ROW_SUCCESS, SQL_ROW_SUCCESS_WITH_INFO,
                   SQL_ROW_ADDED, SQL_ROW_UPDATED];
```

THstmt.SkipByCursor

property SkipByCursor: **Boolean**;

property bool SkipByCursor;

Description

Set this property to True if you want to prevent the statement component from performing cursor operations (Level 2 SQL Generation). The default value is False.

THstmt.SkipByPosition

property SkipByPosition: **Boolean**;

property bool SkipByPosition;

Description

Set this property to True if you want to prevent the statement component from performing positional operations (Level 1 SQL Generation). The default value is False.

THstmt.SQL

property SQL: **String**;

property AnsiString SQL;

Description

Used to set the SQL statement to be executed. The default value is an empty string.

A parameter in the SQL statement can either be unnamed (indicated by a question mark '?') or named (indicated by a single colon ':' followed by a custom parameter name). Single colons ":" are parsed as parameter indicators, unless they occur inside string literals. For example, only "MyFileAttr" is parsed as a parameter in the following SQL statement, while the rest of the colons are ignored:

UPDATE Files SET FileTime = '12:15:45' WHERE FilePath = 'C:\Programs\MyFiles' AND FileAttr = :MyFileAttr

You cannot mix named and unnamed parameters in the same SQL statement. A named parameter can only consist of the characters 'A'..'Z', 'a'..'z', '0'..'9', '_' and '@'.

If a table or column name contains spaces, it must be enclosed with double quotes in the SQL statement, for example:
SQL:= 'SELECT Name, Age, "Identity Number" FROM "Employee Records" WHERE Age <= 30'
Full support for spaces in table and column names are provided by all ODBCExpress components.

ODBCExpress supports both pass-through (database-specific) SQL and ODBC (ANSI-92) SQL. Either of the two or a combination of both can be used in an application. See the ODBC documentation for all the different components of ODBC SQL. The ODBC SQL also supports standard escape clause syntax, which are converted by the ODBC driver to the database-specific syntax. For example, it has the following 3 date/time escape clauses:

{d 'yyyy-mm-dd'} for a date
{t 'hh:mm:ss'} for a time
{ts 'yyyy-mm-dd hh:mm:ss[.f...]'} for a timestamp

Example for 20 March 1980:

UPDATE Students SET SBirth = {d '1980-03-20'} WHERE SNo = ?

Note:

In 16bit Delphi the SQL property is a TStringList instead of a String. This is because strings in 16bit Delphi are limited to 255 characters, while SQL statements are often longer than 255 characters. A long SQL statement can therefore be split across a number of strings, and a split can be at any place where a space can occur in the SQL statement. In 32bit Delphi strings aren't limited to 255 characters - they are virtually of unlimited length.

Example:

Delphi Code

```
var
  SNo, SName: String;
begin
  //students registered in 1997
  SNo:= '97%';
  //students with names starting with 'P'
  SName:= 'P%';

  with Hstmt1 do
  begin
    //set and prepare a SQL statement with 2 unnamed parameters
    SQL:= 'SELECT SNo, SName FROM Students WHERE SNo LIKE ?, SName LIKE ?';
    Prepare;

    //bind the unnamed parameters - can't be bound as named parameters
    BindString(1, SNo);
    BindString(2, SName);

    //execute select statement
    Execute;
  end;

  //or

  with Hstmt1 do
  begin
    //set and prepare a SQL statement with 2 named parameters
    SQL:= 'SELECT SNo, SName FROM Students WHERE SNo LIKE :ANo, SName LIKE
:AName';
    Prepare;

    //bind the named parameters - can also be bound as unnamed parameters
    BindStringByName('ANo', SNo);
    BindString(2, SName);

    //execute select statement
    Execute;
  end;
end;
```

C++ Code

```
AnsiString SNo, SName;
```

```
//students registered in 1997
SNo = "97%";
//students with names starting with 'P'
SName = "P%";

//set and prepare a SQL statement with 2 unnamed parameters
Hstmt1->SQL = "SELECT SNo, SName FROM Students WHERE SNo LIKE ?, SName LIKE ?";
Hstmt1->Prepare();

//bind the unnamed parameters - can't be bound as named parameters
Hstmt1->BindString(1, SNo);
Hstmt1->BindString(2, SName);

//execute select statement
Hstmt1->Execute();

//or

//set and prepare a SQL statement with 2 named parameters
Hstmt1->SQL = "SELECT SNo, SName FROM Students WHERE SNo LIKE :ANo, SName LIKE
:AName";
Hstmt1->Prepare();

//bind the named parameters - can also be bound as unnamed parameters
Hstmt1->BindStringByName("ANo", SNo);
Hstmt1->BindString(2, SName);

//execute select statement
Hstmt1->Execute();
```

THstmt.SQLParsing

property SQLParsing: **Boolean**;

property bool SQLParsing;

Description

Used to switch off parsing of the SQL statement for parameters by setting the property to False. The default value is True.

This is useful for when you concatenate data to your SQL statement and want to prevent the parser from parsing colons as named parameters. Of course setting this property to False means you cannot reference a parameter by name anymore, but only by number.

THstmt.SqlType

property SqlType[Col: SQLUSMALLINT]: SQLSMALLINT;

property SQLSMALLINT SqlType[SQLUSMALLINT Col];

Description

Returns the database data type (data type of column as in database) of column number Col in the current row of the result set. If Col is out of range, it returns 0.

THstmt.StringTrimming

TStringTrimming = (stTrimNone, stTrimTrailing, stTrimLeading, stTrimBoth);

property StringTrimming: TStringTrimming;

TStringTrimming {stTrimNone, stTrimTrailing, stTrimLeading, stTrimBoth};

property TStringTrimming StringTrimming;

Description

Used to control the way in which string data retrieved from a database is trimmed. It can be one of the following values:

stTrimNone	Strings are not trimmed in any way.
stTrimTrailing	Trailing spaces and control characters are trimmed.
stTrimLeading	Leading spaces and control characters are trimmed.
stTrimBoth	Leading and trailing spaces and control characters are trimmed.

The default value is stTrimTrailing.

THstmt.TargetTable

property TargetTable: **String**;

property AnsiString TargetTable;

Description

Set this property when you are making use of Level 2 or 3 SQL Generation and the ODBC Driver you are using cannot determine the name of the table to use in the insert, update or delete operation. Generally it is not necessary to set this property because:

1. Your ODBC Driver might support the level 2 ODBC functionality needed to perform positional operations (Level 1 SQL Generation).
2. Your ODBC Driver might be able to retrieve the name of the table to generate an insert, update or delete statement (Level 2 and 3 SQL Generation).
3. The SQL is parsed to return this information if necessary.

If this property is set manually, it overrides automatic detection of the target table.

THstmt.Terminate

function Terminate: **Boolean**; **override**;

virtual bool Terminate(**void**);

Description

Terminates the association that the statement component has with ODBC. This method is automatically called when the statement component is destroyed.

THstmt.TypeString

function TypeString(SqlType: SQLSMALLINT;
SqlSize: **Longint**): **String**;

AnsiString TypeString(SQLSMALLINT SqlType,
long SqlSize);

Description

A utility method that returns the DataSource dependent data type name corresponding to the database data type SqlType. If SqlSize > 0, it is merged at the correct position with the type name.

Example:

For the Microsoft SQL Server database, the following string values are returned:

TypeString(SQL_INTEGER, 0)	int
TypeString(SQL_CHAR, 0)	char
TypeString(SQL_CHAR, 20)	char(20)

TOEDataSet

Hierarchy Properties Methods Events

Unit ODSI;

Description

The TOEDataSet component provides access to the Delphi and other compliant 3rd-party data-aware controls from ODBCExpress. It implements all the abstract functionality and almost all of the virtual functionality of the virtual TDataSet class. Additionally it provides TQuery-like properties, to allow some level of code compatibility with existing applications written using the TQuery component. Documentation on the Delphi TDataSet properties and methods not specific to the TOEDataSet can be found in the Delphi help file. The following keys describes the non-documented TOEDataSet properties, methods and events:

Supported	Functionality is inherited from the TDataSet or is implemented by the TOEDataSet.
Not Supported	Functionality is not inherited from the TDataSet and is not implemented by the TOEDataSet.
Inherited	Default functionality is inherited from the TDataSet and is not directly implemented by the TOEDataSet.

One of the advantages of the TOEDataSet over existing TDataSet descendants are full ODBC Driver cursor support, which eliminates the need for a resource-heavy front-end cache as used by existing database engines. A cache is used by a database engine which only supports forward-only result set cursors, to allow you to move backwards in the result set via the cache. Cursor-support allows you to move backwards in your result set without having to cache the rows on the front-end. This means scrolling to the last row in your result set takes virtually the same time as scrolling to the first or next row in your result set. Also, since blobs are fetched from the database as they are needed, this means that you can have as many blobs as you want in your result set, no matter what the size of the blobs or how many rows you have in your result set, without having to worry about running out of memory because of a front-end cache. Fetching the last row with existing database engines which makes use of a cache, means that all the rows are fetched to and cached on your application front-end, even the blob fields contained in the rows.

Speed Considerations

Of course, scrollable cursors are slightly slower than forward-only cursors when it comes to SQL databases, but the TOEDataSet supports both types of cursors. This allows you to use the very fast forward-only cursor when you are performing time-critical work, and the slightly slower scrollable cursors when you want to go backward in your result set, typically to allow user browsing of the result set. At this level the difference in speed between scrollable and forward-only cursors are not noticeable. However scrollable cursors gives you the added advantages of scrolling to the end or any other position in the result set in virtually the same time as it takes to scroll to the first record, as well as doing so without the need of a resource-heavy cache. Cursors also allows you to defer blob fetching, which means a blob only needs to be fetched when (and if ever) it is reference. This decreases network traffic and your front-end resource usage. Fetching all the rows (and blobs) to a front-end cache on the other hand definitely does not decrease network traffic.

Implementation

The TOEDataSet consists of all the relevant properties and methods of the virtual TDataSet, the virtualized TFields class containing all the TField classes of the TOEDataset, the TOEParams class containing all the TOEParam classes of the TOEDataSet, and a TDataSetHstmt class which contains a subset functionality of the THstmt statement component and which forms the core of the TOEDataSet functionality. A custom TOEBlobStream class, which is mostly used internally by the TOEDataSet, is used to access blobs contained in the TOEDataSet.

The TOEDataSet is completely SQL driven. It is good practice to filter your result set using your SQL statement and implement calculated fields as part of your SQL statement as far as possible. Complex Locates and Lookups negatively influence performance, and it's therefore also good practice to make use of your SQL statement to achieve this where possible.

The following field and parameter data types are directly supported by the TOEDataSet:

ftString	TStringField
ftMemo	TMemoField
ftBytes	TBytesField
ftBlob	TBlobField

ftBoolean	TBooleanField
ftFloat	TFloatField
ftCurrency	TCurrencyField
ftSmallint	TSmallintField
ftWord	TWordField
ftInteger	TIntegerField

ftDate	TDateField
ftTime	TTimeField
ftDateTime	TDateTimeField

TOEDataSet.AbortQuery

procedure AbortQuery;

void AbortQuery(**void**);

Description

Used to cancel the processing of a query executed asynchronously. See the ExecAsync property for an example.

TOEDataSet.ApplyUpdates

procedure ApplyUpdates(UpdateStatus: TUpdateStatusSet);

procedure ApplyUpdates;

void ApplyUpdates(TUpdateStatusSet UpdateStatus);

void ApplyUpdates(**void**);

Description

These two overloaded methods allows you to apply either all the cached updates (inserts, modifications and deletes) or a subset of the cached updates.

This method will automatically clear the updated records from the cache (similar to calling the CommitUpdates method in the BDE) only after the updates have been successfully applied. If an exception is raised during the update process (and it's not handled by the OnUpdateError event), the updated records will not be cleared from the cache. This allows you to safely roll back the transaction if the ApplyUpdates method fails, without losing any of your updates.

TOEDataSet.Cached

property Cached: **Boolean**;

property bool Cached;

Description

This property is used to set whether the result set must be cached on the front-end (True), or whether a back-end result set must be used (False). The default value is False.

When the result set is cached on the front end, only a read-only concurrency type and forward-only cursor type is needed (though you can still make use of the other values). Level 1 and Level 2 SQL Generation cannot be used together with a front-end cache, so the SkipByPosition and SkipByCursor sub-properties of the Hstmt property will automatically be set to True.

TOEDataSet.CachedUpdates

property CachedUpdates: **Boolean**;

property bool CachedUpdates;

Description

This property is used to set whether update operations (inserts, modifications and deletes) must be cached to be applied at a later stage (True) or send through to the database immediately (False). The default value is False.

Setting it to True will automatically set the Cached property to True as well, since CachedUpdates can only be used together with a front-end cache. The ApplyUpdates, CancelUpdates, FlushBuffers and ReverdRecord methods and the UpdatesPending and UpdateStatus properties can be used together with cached updates. Though no UpdateMode property is provided, the update mode is key-only by default and can be changed to all the fields or a subset of the fields using the Target sub-property of the Hstmt property.

TOEDataSet.CancelUpdates

procedure CancelUpdates(UpdateStatus: TUpdateStatusSet);

procedure CancelUpdates;

void CancelUpdates(TUpdateStatusSet UpdateStatus);

void CancelUpdates(**void**);

Description

These two overloaded methods allows you to cancel either all the cached updates (inserts, modifications and deletes) or a subset of the cached updates.

TOEDataSet.CreateHstmt

function CreateHstmt: THstmt; **virtual**;

virtual THstmt* CreateHstmt;

Description

The THstmt contained by the TOEDataSet is created via this protected virtual method. You can override and return a customized THstmt descendant component from it, without calling the inherited method which creates an original THstmt component by default.

TOEDataSet.DataSource

property DataSource: TDataSource;

property TDataSource DataSource;

Description

Allows the easy creation of master-detail relationships between TOEDataSet components. This property must be set in the detail dataset component to the DataSource which points to the master dataset component from which to extract current field values to use to bind otherwise unassigned parameters in the query SQL statement.

Note:

This property must not be confused with the THdbc.DataSource property, which indicates the database connected to via an ODBC DataSource.

TOEDataSet.DoDelete

function DoDelete(Sender: TObject;
DefaultMsg: **String**): **Boolean**; **virtual**;

virtual bool DoDelete(TObject* Sender,
AnsiString DefaultMsg);

Description

Calls the OnUpdateRecord event with an UpdateKind of ukDelete.

TOEDataSet.DoInsert

function DoInsert(Sender: TObject;
DefaultMsg: **String**): **Boolean**; **virtual**;

virtual bool DoInsert(TObject* Sender,
AnsiString DefaultMsg);

Description

Calls the OnUpdateRecord event with an UpdateKind of ukInsert.

TOEDataSet.DoRefresh

function DoRefresh(Sender: TObject;
DefaultMsg: **String**): **Boolean**; **virtual**;

virtual bool DoRefresh(TObject* Sender,
AnsiString DefaultMsg);

Description

Calls the OnRefreshRecord event.

TOEDataSet.DoUpdate

function DoUpdate(Sender: TObject;
DefaultMsg: **String**): **Boolean**; **virtual**;

```
virtual bool DoUpdate(TObject* Sender,
                      AnsiString DefaultMsg);
```

Description

Calls the OnUpdateRecord event with an UpdateKind of ukModify.

TOEDataSet.DoUpdateError

```
procedure DoUpdateError(E: EODBC;
                        UpdateKind: TUpdateKind;
                        var RaiseError: Boolean); virtual;
```

```
virtual void DoUpdateError(EODBC* E,
                           TObject* Sender,
                           bool &RaiseError);
```

Description

Calls the OnUpdateError event.

TOEDataSet.DoUpdateRecord

```
procedure DoUpdateRecord(UpdateKind: TUpdateKind;
                         var ApplyUpdate: Boolean); virtual;
```

```
virtual void DoUpdateRecord(TObject* Sender,
                            bool &ApplyUpdate);
```

Description

Calls the OnUpdateRecord event.

TOEDataSet.Editable

```
property Editable: Boolean;
```

```
property bool Editable;
```

Description

This property is used to set whether the data-aware controls pointing to the TOEDataSet are editable or not. The default value is False. It is similar to the RequestLive property of the TQuery, however when set to True it will always return a potentially modifiable result set if the user has the appropriate rights. Whether actual inserts. updates, deletes, etc. can be performed depends on the Level of SQL Generation used. If set to True and an updatable concurrency is required, the concurrency is changed accordingly.

TOEDataSet.EmptyFieldToNull

property EmptyFieldToNull: **Boolean**;

property bool EmptyFieldToNull;

Description

Allows you to control the way empty fields are treated during positional insert operations. If True, empty fields are inserted with null values at the database. If False, provision is made for usage of the default values (as set up at the database) for empty fields. The default value of this property is False.

TOEDataSet.ExecSQL

procedure ExecSQL;

void ExecSQL(**void**);

Description

This method is used to execute a SQL statement which does not return a result set

TOEDataSet.ExtendedState

TOEDataSetState = (dsNotExtended, dsLocate, dsCachedUpdates);

property ExtendedState: TOEDataSetState;

TOEDataSetState {dsNotExtended, dsLocate, dsCachedUpdates};

property TOEDataSetState ExtendedState;

Description

This property is an extension of the State property of the TDataSet and adds some TOEDataSet-specific states. The default value is dsNotExtended. The dsNotExtended value indicates that the value of the State property applies.

TOEDataSet.FetchAll

procedure FetchAll;

void FetchAll(**void**);

Description

This method will fill the front-end cache with all the records in the result set.

TOEDataSet.FlushBuffers

procedure FlushBuffers;

void FlushBuffers(**void**);

Description

Used to cause the dataset to apply all cached updates (inserts, modifications and deletes). This method is used instead of the CheckBrowseMode method if it is important that the cached updates must be applied.

TOEDataSet.Hdbc

property hDbc: THdbc;

property THdbc* hDbc;

Description

This property is set to the connection component which identifies the database to be accessed by the TOEDataSet.

TOEDataSet.Hstmt

property hStmt: TDataSetHstmt;

property TDataSetHstmt* hStmt;

Description

This property contains a subset functionality of the THstmt statement component, by exposing only the properties and methods of the statement component relevant to the TOEDataSet component.

TOEDataSet.LocateInsert

property LocateInsert: **Boolean**;

property bool LocateInsert;

Description

Used to avoid the "disappearing inserts" affect in the TOEDataSet. A row inserted into a result set has no position in the result set, causing the row to disappear from view when using visual data-aware controls. When this property is set to True, it makes use of the target primary key values (which are generally automatically detected) to close and re-open the dataset and then position to the inserted row (if it matches the search criteria of the new result set). The default value is False.

This property also allows you to force inserts to comply with the sort order when making use of the cache.

TOEDataSet.LocateOption

TOELocateOption = (loPrefix, loSubString);

property LocateOption: TOELocateOption;

TOELocateOption {loPrefix, loSubString};

property TOELocateOption LocateOption;

Description

When doing partial string matching with the Locate method, this property can be used to specify whether to do prefix or sub-string partial matching. The default value is loPrefix.

TOEDataSet.MainHstmt

property MainHstmt: THstmt;

property THstmt* MainHstmt;

Description

This protected property provides access to the actual THstmt component contained by the TOEDataSet.

TOEDataSet.MoveAbsolute

function MoveAbsolute(Distance: **Integer**): **Boolean**;

bool MoveAbsolute(**int** Distance);

Description

Used to perform pure cursor-based absolute row fetching, which is not possible using the standard MoveBy method. A positive value will fetch absolute from the beginning of the result set and a negative value will fetch absolute from the end of the result set.

TOEDataSet.MoveRelative

function MoveRelative(Distance: **Integer**): **Boolean**;

bool MoveRelative(**int** Distance);

Description

Used to perform pure cursor-based relative row fetching, which is not possible using the standard MoveBy method. A positive value will fetch forward from the current row in the result set and a negative value will fetch backward from the current row in the result set.

TOEDataSet.NextResultSet

function NextResultSet: **Boolean**;

bool NextResultSet(**void**);

Description

This method will initialize processing for for the next result set when multiple result sets are returned by the TOEDataSet. It will close and re-open the DataSet to display the next result set.

This method also provides a way for completing the processing of stored procedures executed with some ODBC drivers so that the return and result values of the stored procedures can be accessed.

Example:

Delphi Code

```
with OEDataSet1 do
begin
  if not Active then
  begin
    //open first result set
    SQL:= 'SELECT SNo, SName, SGender FROM Students; '+
          'SELECT CCode, CDescr FROM Courses; '+
          'SELECT * FROM Takes';
    Open;
  end
  //open next result set
  else if not NextResultSet then
    ShowMessage('No More Results');
end;
```

C++ Code

```
if (!OEDataSet1->Active) {
  //open first result set
  OEDataSet1->SQL = "SELECT SNo, SName, SGender FROM Students; \
                     SELECT CCode, CDescr FROM Courses; \
                     SELECT * FROM Takes";
  OEDataSet1->Open();
}
//open next result set
else if (!OEDataSet1->NextResultSet()) {
  ShowMessage("No More Results");
}
```

TOEDataSet.OnRefreshRecord

property OnRefreshRecord: TNotifyEvent;

property TNotifyEvent OnRefreshRecord;

Description

This event is called on each Refresh of the current row made with the DataSet component.

TOEDataSet.OnUpdateError

```
TOEUpdateErrorEvent = procedure (DataSet: TDataSet;
                                 E: EODBC;
                                 UpdateKind: TUpdateKind;
                                 var RaiseError: Boolean) of object;
```

```
property OnUpdateError: TOEUpdateErrorEvent;
```

```
void (closure* TOEUpdateErrorEvent)(TDataSet* DataSet,
                                    EODBC* E,
                                    TObject* Sender,
                                    bool &RaiseError);
```

```
property TOEUpdateErrorEvent OnUpdateError;
```

Description

This event is called when a cached update (modification, insert or delete) in the DataSet cannot be applied. UpdateKind specifies the type of update operation (ukModify, ukInsert or ukDelete) which failed. E is the ODBC exception which occured and RaiseError is used to indicate whether the exception must be raised or not. The default value of RaiseError is True.

If the exception is raised, the update process will be cancelled and the current and following cached updates will remain unchanged in the cache, while the preceding cached updates will have been applied. If the exception is not raised, the current cached update will be lost (unless handled inside the event) and the update process will continue with the following cached updates.

Modifications are applied from the start to the end of the cached, while inserts and deletes are applied from the end to the start of the cache.

TOEDataSet.OnUpdateRecord

```
TOEUpdateRecordEvent = procedure (DataSet: TDataSet;
                                  UpdateKind: TUpdateKind;
                                  var ApplyUpdate: Boolean) of object;
```

```
property OnUpdateRecord: TOEUpdateRecordEvent;
```

```
void (closure* TOEUpdateRecordEvent)(TDataSet* DataSet,
                                     TObject* Sender,
                                     bool &ApplyUpdate);
```

```
property TOEUpdateRecordEvent OnUpdateRecord;
```

Description

This event is called before an update operation (modification, insert or delete) occurs in the DataSet. UpdateKind specifies the type of update operation (ukModify, ukInsert or ukDelete) and

ApplyUpdate is used to indicate whether the update operation must continue or not. The default value of ApplyUpdate is True.

TOEDataSet.ParamByName

function ParamByName(ParamName: **String**): TParam;

TParam* ParamByName(**AnsiString** ParamName);

Description

This method is used to return the Delphi TParam parameter object associated with the parameter ParamName in the SQL statement.

TOEDataSet.ParamCheck

property ParamCheck: **Boolean**;

property bool ParamCheck;

Description

This property is used to determine if the parameter objects must be re-generated when the SQL statement is changed at run-time. Set to False to avoid the re-generation of the parameter objects. The default value is True.

TOEDataSet.ParamCount

property ParamCount: **Word**;

property Word ParamCount;

Description

Returns the number of parameters in the SQL statement.

TOEDataSet.ParamNames

property ParamNames: TStrings;

property TStrings* ParamNames;

Description

Returns a list of the parameter names of the currently prepared SQL statement.

TOEDataSet.Params

property Params: TOEParams;

property TOEParams* Params;

Description

Returns the parameters object associated with the SQL statement.

TOEDataSet.Prepare

procedure Prepare;

void Prepare(**void**);

Description

This method is used to prepare the SQL statement. It is automatically called when the TOEDataSet is opened using the Open method, or when ExecSQL is called.

TOEDataSet.Prepared

property Prepared: **Boolean**;

property bool Prepared;

Description

This property is used to check if the statement in the TOEDataSet is prepared or not. It can also be used to prepare or unprepare the statement:

Prepared:= True is equivalent to Prepare.
Prepared:= False is equivalent to UnPrepare.

TOEDataSet.RevertRecord

procedure RevertRecord;

void RevertRecord(**void**);

Description

Used to undo the pending cached update of the current selected row in the DataSet. A pending inserted record will be discarded and a pending modified record will be changed back to its unmodified state. Pending deleted records can only be reverted by calling one of the overloaded CancelUpdates methods.

TOEDataSet.RowsAffected

property RowsAffected: **Integer**;

property int RowsAffected;

Description

Returns the number of rows affected by an insert, update or delete statement.

TOEDataSet.StoredProc

property StoredProc: **String**;

property AnsiString StoredProc;

Description

This is an alternative to the SQL and Table properties. When this property is set to indicate a stored procedure at the database, the result set of the stored procedure will be returned to the TOEDataSet. The Params property will also automatically be filled with the stored procedure's parameters, if possible. It has precedence over the SQL property and when set will clear the Table property.

The property editor, which can be accessed via the Object Inspector, lists all the stored procedures available at the current DataSource.

TOEDataSet.SQL

property SQL: **String**;

property AnsiString SQL;

Description

This is the SQL statement used by the TOEDataSet to return a result set, or perform inserts, updates and deletes. When this property is set, the Table and StoredProc properties are automatically cleared to remove the precedence that the Table and StoredProc properties has over this property.

The property editor, which can be accessed via the Object Inspector, contains a SQL Builder.

TOEDataSet.Table

property Table: **String**;

property AnsiString Table;

Description

This is an alternative to the SQL and StoredProc properties. When this property is set to indicate a table at the database, all the records contained in the table will be included in the result set of the TOEDataSet. It has precedence over the SQL property and when set will clear the StoredProc property.

The property editor, which can be accessed via the Object Inspector, lists all the tables and views available at the current DataSource.

TOEDataSet.UnBind

procedure UnBind;

void UnBind(**void**);

Description

This method is used to un-bind the parameters of the dataset. This will cause the parameters to be re-bound to the DataSource the next time the dataset is opened or executed. This method is mostly used internally by the dataset.

TOEDataSet.UnPrepare

procedure UnPrepare;

void UnPrepare(**void**);

Description

This method is used to indicate the SQL statement has to be re-prepared the next time the Prepare method is called.

TOEDataSet.UpdateObject

property UpdateObject: TOEUpdateObject;

property TOEUpdateObject* UpdateObject;

Description

Used to set the UpdateObject component to perform positional insert, update and delete operations (level 4 SQL Generation). When set, this overrides the first three levels of SQL Generation. This property can be used instead of the OnInsert, OnUpdate and OnDelete events to perform the level 4 SQL Generation, though these events can still be used to confirm these operations.

TOEDataSet.UpdatesPending

property UpdatesPending: **Boolean**;

property bool UpdatesPending;

Description

Used to check whether there are any cached updates (inserts, modifications and deletes) which have not been written to the database yet.

TDataSetHstmt

Hierarchy Properties Methods

Unit ODSI;

Description

This class contains a subset functionality of the THstmt statement component, by exposing only the properties and methods of the statement component relevant to the TOEDataSet component.

TDataSetHstmt.BindBookmarks

property BindBookmarks: **Boolean**;

property bool BindBookmarks;

Description

Performs the same functionality as the BindBookmarks property of the statement component. The default value is False.

TDataSetHstmt.BindByName

property BindByName: **Boolean**;

property bool BindByName;

Description

Performs the same functionality as the BindByName property of the statement component. The default value is False.

TDataSetHstmt.BlobDeferral

property BlobDeferral: **Boolean**;

property bool BlobDeferral;

Description

Performs the same functionality as the BlobDeferral property of the statement component. The default value is False.

This is equivalent to the functionality controlled by the CacheBlobs property of the BDE's TQuery component.

TDataSetHstmt.BlobPlacement

property BlobPlacement: TBlobPlacement;

property TBlobPlacement BlobPlacement;

Description

Performs the same functionality as the BlobPlacement property of the statement component. The default value is bpDetect.

TDataSetHstmt.BlobSize

property BlobSize: **Longint**;

property long BlobSize;

Description

Performs the same functionality as the BlobSize property of the statement component. The default value is False.

TDataSetHstmt.ConcurrencyType

property ConcurrencyType: SQLUINTEGER;

property SQLUINTEGER ConcurrencyType;

Description

Performs the same functionality as the ConcurrencyType property of the statement component. The default value is SQL_CONCUR_READ_ONLY.

TDataSetHstmt.CursorType

property CursorType: SQLUINTEGER;

property SQLUINTEGER CursorType;

Description

Performs the same functionality as the CursorType property of the statement component. The default value is SQL_CURSOR_KEYSET_DRIVEN.

TDataSetHstmt.EmptyStringToNull

TEmptyToNull = (enNever, enAlways, enIfNullable);

property EmptyStringToNull: TEmptyToNull;

TEmptyToNull {enNever, enAlways, enIfNullable};

property TEmptyToNull EmptyStringToNull;

Description

Performs the same functionality as the EmptyStringToNull property of the statement component. The default value is enNever.

TDataSetHstmt.ExecAsync

property ExecAsync: **Boolean**;

property bool ExecAsync;

Description

Allows you to make use of cancelable queries. The default value is False. Setting this property to True will enable asynchronous execution. The Open method, though still a blocking call, will then process application messages to allow you to call the AbortQuery method from the same (or different) thread to cancel query processing. Calling the AbortQuery method will cause the Open method to return by raising the silent EABORT exception. If the AbortQuery method is not called the Open method will return normally when query processing has completed.

Example:

Delphi Code

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  //enable asynchronous execution
  OEDataSet1.Hstmt.ExecAsync:= True;

  //open time-consuming query
  OEDataSet1.SQL:= 'SELECT * FROM Students WHERE SComment LIKE ?';
  OEDataSet1.Open;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  //cancel query processing
  OEDataSet1.AbortQuery;
end;
```

C++ Code

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  //enable asynchronous execution
  OEDataSet1->Hstmt->ExecAsync = true;

  //open time-consuming query
  OEDataSet1->SQL = "SELECT * FROM Students";
  OEDataSet1->Open();
}

void __fastcall TForm1::Button2Click(TObject *Sender)
{
  //cancel query processing
  OEDataSet1->AbortQuery();
}
```

TDataSetHstmt.DataSet

property DataSet: TOEDataSet;

property TOEDataSet DataSet;

Description

This returns the TOEDataSet component associated with the class.

TDataSetHstmt.Handle

function Handle: SQLHSTMT;

SQLHSTMT Handle(**void**);

Description

Performs the same functionality as the Handle method of the statement component.

TDataSetHstmt.Hstmt

property hStmt: THstmt;

property THstmt* hStmt;

Description

This returns the actual THstmt statement component contained in the class.

TDataSetHstmt.MaxRows

property MaxRows: SQLUINTEGER;

property SQLUINTEGER MaxRows;

Description

Performs the same functionality as the MaxRows property of the statement component.

TDataSetHstmt.NoRowsAffected

type TNoRowsAffected = **set of** (nrByInsert, nrByUpdate, nrByDelete, nrByRefresh);

property NoRowsAffected: TNoRowsAffected;

enum temp {nrByInsert, nrByUpdate, nrByDelete, nrByRefresh};
typedef Set<temp, nrByInsert, nrByRefresh> TNoRowsAffected;

property TNoRowsAffected NoRowsAffected;

Description

Performs the same functionality NoRowsAffected as the property of the statement component. By default the set only contains the value nrByUpdate.

When used together with cached updates, the OnUpdateError event will be called if implemented. In this case, as well as any other time when using level 3 SQL Generation, it tries to find a row to update based on the primary columns (which might be set using the Target property). In other words you can simulate some of the BDE update mode property values as follows:

upWhereAll	Mark all the fields as primary using the target property.
upWhereChanged	Not possible to simulate this value.
upWhereKeyOnly	Mark only the primary key fields as primary using the target property (or leave it up to the primary key detection functionality).

TDataSetHstmt.QueryTimeOut

property QueryTimeOut: SQLUINTEGER;

property SQLUINTEGER QueryTimeOut;

Description

Performs the same functionality as the QueryTimeOut property of the statement component.

TDataSetHstmt.RowCountMethod

TRowCountMethod = (rcFunction, rcSelect, rcTraverse, rcCustom);

property RowCountMethod: TRowCountMethod;

TRowCountMethod {rcFunction, rcSelect, rcTraverse, rcCustom};

property TRowCountMethod RowCountMethod;

Description

Performs the same functionality as the RowCountMethod property of the statement component. The default value is rcFunction.

In a cached DataSet, the rcCustom option fills the cache with all the rows in the result set and then returns the number of rows in the cache.

TDataSetHstmt.RowSetSize

property RowSetSize: SQLUINTEGER;

property SQLUINTEGER RowSetSize;

Description

Performs the same functionality as the RowSetSize property of the statement component. The default value is False.

A value larger than 1 can only be used together with a front-end cache in the TOEDataSet to retrieve multiple rows from the back-end at a time into the cache. This is implemented transparently and does not affect the use of the DataSet's fetch methods.

TDataSetHstmt.SkipByCursor

property SkipByCursor: **Boolean**;

property bool SkipByCursor;

Description

Performs the same functionality as the SkipByCursor property of the statement component. The default value is False.

TDataSetHstmt.SkipByPosition

property SkipByPosition: **Boolean**;

property bool SkipByPosition;

Description

Performs the same functionality as the SkipByPosition property of the statement component. The default value is False.

TDataSetHstmt.SQLParsing

property SQLParsing: **Boolean**;

property bool SQLParsing;

Description

Performs the same functionality as the SQLParsing property of the statement component. The default value is True.

TDataSetHstmt.StringTrimming

TStringTrimming = (stNone, stTrimTrailing, stTrimLeading, stTrimBoth);

property StringTrimming: TStringTrimming;

TStringTrimming {stNone, stTrimTrailing, stTrimLeading, stTrimBoth};

property TStringTrimming StringTrimming;

Description

Performs the same functionality as the StringTrimming property of the statement component. The default value is stTrimTrailing.

TDataSetHstmt.Target

property Target: TTarget;

property TTarget* Target;

Description

Used to specify the target table, primary columns and columns to be ignored for Levels 2 and 3 SQL Generation.

A property editor, which can be accessed via the Object Inspector, exists for this property. This property editor can automatically detect the target table, primary and ignore columns if necessary.

TTarget

Hierarchy Properties Methods

Unit ODSI;

Description

This class is used to specify the target table, primary columns and columns to be ignored for Levels 2 and 3 SQL Generation with the TOEDataSet.

TTarget.Clear

procedure Clear;

void Clear(**void**);

Description

Allows you to clear the TargetTable, PrimaryCols and IgnoreCols fields of the TTarget sub-property.

TTarget.IgnoreCols

procedure IgnoreCols(Cols: **array of String**;
ColCount: **Integer**);

```
procedure IgnoreCols(Cols: array of String);

void IgnoreCols(AnsiString* Cols,
                const int Cols_Size,
                int ColCount);

void IgnoreCols(AnsiString* Cols,
                const int Cols_Size);
```

Description

Indicates the columns in a result set to be ignored during positional Inserts and Updates. Calling this method is equivalent to calling both the IgnoreInsertCols and IgnoreUpdateCols methods with the same parameters. See the IgnoreCols method of the statement component for more details.

TTarget.IgnoreInsertColNames

```
property IgnoreInsertColNames: TStringList;

property TStringList* IgnoreInsertColNames;
```

Description

This allows you to read the existing columns to be ignored during positional inserts, as well as add or remove individual columns to the list. The Clear method can be used to remove all the columns from the list.

TTarget.IgnoreInsertCols

```
procedure IgnoreInsertCols(Cols: array of String;
                           ColCount: Integer);

procedure IgnoreInsertCols(Cols: array of String);

void IgnoreInsertCols(AnsiString* Cols,
                      const int Cols_Size,
                      int ColCount);

void IgnoreInsertCols(AnsiString* Cols,
                      const int Cols_Size);
```

Description

Indicates the columns in a result set to be ignored during positional Inserts. See the IgnoreCols method of the statement component for more details.

TTarget.IgnoreUpdateColNames

```
property IgnoreUpdateColNames: TStringList;

property TStringList* IgnoreUpdateColNames;
```

Description

This allows you to read the existing columns to be ignored during positional updates, as well as add or remove individual columns to the list. The Clear method can be used to remove all the columns from the list.

TTarget.IgnoreUpdateCols

procedure IgnoreUpdateCols(Cols: **array of String**;
                                        ColCount: **Integer**);

procedure IgnoreUpdateCols(Cols: **array of String**);

void IgnoreUpdateCols(**AnsiString*** Cols,
                              const int Cols_Size,
                              int ColCount);

void IgnoreUpdateCols(**AnsiString*** Cols,
                              const int Cols_Size);

Description

Indicates the columns in a result set to be ignored during positional Updates. See the IgnoreCols method of the statement component for more details.

TTarget.PrimaryColNames

property PrimaryColNames: TStringList;

property TStringList* PrimaryColNames;

Description

This allows you to read the existing primary columns, as well as add or remove individual columns to the list. The Clear method can be used to remove all the columns from the list.

TTarget.PrimaryCols

procedure PrimaryCols(Cols: **array of String**;
                                 ColCount: **Integer**);

procedure PrimaryCols(Cols: **array of String**);

void PrimaryCols(**AnsiString*** Cols,
                        const int Cols_Size,
                        int ColCount);

void PrimaryCols(**AnsiString*** Cols,
                       const int Cols_Size);

Description

Performs the same functionality as the PrimaryCols method of the statement component.

TTarget.TargetTable

property TargetTable: **String**;

property AnsiString TargetTable;

Description

Performs the same functionality as the TargetTable property of the statement component.

TOEParams

Hierarchy Properties

Unit ODSI;

Description

The TOEParams class is derived from the Delphi TParams class and adds some additional properties. It supports all the TParams properties and methods, including the TParam properties and methods.

TOEParams.DefDataType

property DefDataType[Param: **String**]: TFieldType;

property TFieldType DefDataType[**AnsiString** Param];

Description

Indicates the default data type of the parameter Param. This is a read-only property.

TOEParams.DefPrecision

property DefPrecision[Param: **String**]: SQLSMALLINT;

property SQLSMALLINT DefPrecision[**AnsiString** Param];

Description

Indicates the scale of the parameter Param. This is a read-only property.

TOEParams.DefRequired

property DefRequired[Param: **String**]: **Boolean**;

property bool DefRequired[**AnsiString** Param];

Description

Indicates whether the parameter Param is by default nullable or not. This is a read-only property.

TOEParams.DefSize

property DefSize[Param: **String**]: SQLUINTEGER;

property SQLUINTEGER DefSize[**AnsiString** Param];

Description

Indicates the defined size of the parameter Param. This is a read-only property.

TOEBlobStream

Hierarchy Properties Methods

Unit ODSI;

Description

This class is used to manipulate a blob column contained in a TOEDataSet. It is normally only used internally by the TBlobField and descendant field classes, and is similar to the TBlobStream class.

TOEBlobStream.MemoryStream

property MemoryStream: TMemoryStream;

property TMemoryStream* MemoryStream;

Description

This property provides access to the TMemoryStream object which contains the actual value of the blob field. It's rarely necessary to access this property, since the properties and methods of the TOEBlobStream class is rather used to manipulate the blob field.

TOEBlobStream.Modified

procedure Modified;

void Modified(void);

Description

This method is used to notify the TOEDataSet that the Blob Field contained by this TOEBlobStream class has been modified. It only needs to be called when modifying the MemoryStream property (described below) directly, instead of making use of the TOEBlobStream properties and methods.

TOEBlobStream.Read

function Read(**var** Buffer;
Count: **Longint**): **Longint**; **override;**

virtual long Read(void* Buffer,
long Count);

Description

This method is used to read Count bytes from the TOEBlobStream into Buffer.

TOEBlobStream.Seek

function Seek(Offset: **Longint**;
Origin: **Word**): **Longint**; **override**;

virtual long Seek(long Offset,
Word Origin);

Description

This method is used to position the TOEBlobStream cursor at position Origin+Offset in the stream.

TOEBlobStream.Write

function Write(**const** Buffer;
Count: **Longint**): **Longint**; **override;**

virtual long Write(const void* Buffer,
long Count);

Description

This method is used to write Count bytes from the Buffer into the TOEBlobStream.

TOEQuery

Hierarchy Properties

Unit ODSI;

Description

The TOEQuery component adds more TQuery-like properties to the TCustomOEDataSet, thereby providing a component which will be very familiar to the BDE TQuery users. This component attemps to ease the migration to ODBCExpress and helps to easily convert existing projects. It also incorporates appropriate functionality from the BDE's TTable and TStoredProc components.

TOEQuery.CacheBlobs

property CacheBlobs: **Boolean**;

property bool CacheBlobs;

Description

This property is a TQuery-like alternative to the BlobDeferral sub-property of the hStmt property.

TOEQuery.ParamBindMode

TOEParamBindMode = (pbByName, pbByNumber);

property ParamBindMode: TOEParamBindMode;

TOEParamBindMode {pbByName, pbByNumber};

property TOEParamBindMode ParamBindMode;

Description

This property is a TQuery-like alternative to the BindByName sub-property of the hStmt property.

TOEQuery.RequestLive

property RequestLive: **Boolean**;

property bool RequestLive;

Description

This property sets the Hstmt.ConcurrencyType sub-property to SQL_CONCUR_VALUES if True and to SQL_CONCUR_READ_ONLY if False. The default value is False.

TOEQuery.SQL

property SQL: TStrings;

property TStrings* SQL;

Description

This property is the same as the TOEDataSet.SQL property, except that it is of type TStrings instead of type String. A SQL statement can be split across lines in the TStrings object at any place where a space can occur in the SQL statement.

TOEQuery.StoredProcName

property StoredProcName: **String**;

property AnsiString StoredProcName;

Description

This property is the same as the TOEDataSet.StoredProc property, and is provided to migrate BDE TStoredProc components to TOEQuery components.

TOEQuery.TableName

property TableName: **String**;

property AnsiString TableName;

Description

This property is the same as the TOEDataSet.Table property, and is provided to migrate BDE TTable components to TOEQuery components.

TOEQuery.UniDirectional

property UniDirectional: **Boolean**;

property bool UniDirectional;

Description

This property sets the Hstmt.CursorType sub-property to SQL_CURSOR_FORWARD_ONLY if True and to SQL_CURSOR STATIC or SQL_CURSOR_KEYSET_DRIVEN if False, depending on whether the cursor library is used or not respectively. The default value is False.

TOECatalog

Hierarchy Properties Methods

Unit OVCL;

Description

The TOECatalog Non-Visual Component is used to retrieve catalog information from a database such as tables, stored procedures and their columns and parameters.

TOECatalog.Hdbc

property hDbc: THdbc;

property THdbc* hDbc;

Description

This property is set to the connection component which identifies the database from which the catalog information must be retrieved.

TOECatalog.ParseForeignKey

procedure ParseForeignKey(ForeignKey: **String**;
                                        var ColumnName, ForeignOwner, ForeignTable, ForeignColumn:
String);

void ParseForeignKey(**AnsiString** ForeignKey,
                              AnsiString &ColumnName,
                              AnsiString &ForeignOwner,
                              AnsiString &ForeignTable,
                              AnsiString &ForeignColumn);

Description

Parses the foreign key strings returned by the TCatalogTable.ForeignKeys property into its sub-components.

TOECatalog.ParseIndex

procedure ParseIndex(Index: **String**;
                              var IndexName: **String**;
                              var Unique: **Boolean**;
                              ColumnNames: TStrings);

void ParseIndex(**AnsiString** Index,
                        AnsiString &IndexName,
                        bool &Unique,
                        TStrings* ColumnNames);

Description

Parses the index strings returned by the TCatalogTable.Indexes property into its sub-components.

TOECatalog.ParseUniqueKey

procedure ParseUniqueKey(UniqueKey: **String**;
                                      ColumnNames: TStrings);

void ParseUniqueKey(**AnsiString** UniqueKey;
                              TStrings* ColumnNames);

Description

Parses the unique key string returned by the TCatalogTable.UniqueKey property into its sub-components.

TOECatalog.ProcedureByName

function ProcedureByName(AProcedureOwner, AProcedureName: **String**): TCatalogProcedure;

TCatalogProcedure* ProcedureByName(**AnsiString** AProcedureOwner,
AnsiString AProcedureName);

Description

This method returns the procedure object TCatalogProcedure which corresponds to the procedure name, as retrieved using the Procedures and ProcedureNames properties.

TOECatalog.ProcedureName

property ProcedureName: **String**;

property AnsiString ProcedureName;

Description

This property contains a mask used to limit the procedures retrieved from the DataSource. The default value is a percentage character %, which indicates all the procedures of the specified users must be retrieved. The mask can contain the following meta-characters:

"_"	Underscore meta-character represents any single character.
"%"	Percent meta-character represents any sequence of zero or more characters.
"\"	Escape meta-character which permits the underscore, percent and escape meta-characters to be used as literal characters in the mask.

Example:

B_	Return all procedures that start with a "B" and are two characters long.
B%	Return all procedures that start with a B.
%_%	Return all procedures that contain an underscore character.

TOECatalog.ProcedureNames

property ProcedureNames: TStrings;

property TStrings* ProcedureNames;

Description

This property returns the list of stored procedure names at the DataSource as identified by the Hdbc property.

TOECatalog.ProcedureOwner

property ProcedureOwner: **String**;

property AnsiString ProcedureOwner;

Description

This property contains a mask used to limit the procedures retrieved from the DataSource. The default value is an empty string, which indicates the specified procedures of all users must be retrieved. The mask can contain the following meta-characters:

"_" Underscore meta-character represents any single character.
"%" Percent meta-character represents any sequence of zero or more characters.
"\" Escape meta-character which permits the underscore, percent and escape meta-characters to be used as literal characters in the mask.

Example:

B_	Return all procedures with owners that start with a "B" and are two characters long.
B%	Return all procedures with owners that start with a B.
%_%	Return all procedures with owners that contain an underscore character.

TOECatalog.Procedures

property Procedures: TCatalogProcedures;

property TCatalogProcedures* Procedures;

Description

This property returns the list of stored procedures at the DataSource as identified by the Hdbc property as TCatalogProcedure objects.

TOECatalog.Refresh

procedure Refresh;

void Refresh(**void**);

Description

This method is used to refresh the values contained by the Tables, Procedures, TableNames and ProcedureNames properties.

TOECatalog.TableByName

function TableByName(ATableOwner, ATableName: **String**): TCatalogTable;

TCatalogTable* TableByName(**AnsiString** ATableOwner,
AnsiString ATableName);

Description

This method returns the table object TCatalogTable which corresponds to the table name, as retrieved using the Tables and TableNames properties.

TOECatalog.TableName

property TableName: **String**;

property AnsiString TableName;

Description

This property contains a mask used to limit the tables retrieved from the DataSource. The default value is a percentage character %, which indicates all the tables of the specified users must be retrieved. The mask can contain the following meta-characters:

"_"	Underscore meta-character represents any single character.
"%"	Percent meta-character represents any sequence of zero or more characters.
"\"	Escape meta-character which permits the underscore, percent and escape meta-characters to be used as literal characters in the mask.

Example:

B_	Return all tables that start with a "B" and are two characters long.
B%	Return all tables that start with a B.
%_%	Return all tables that contain an underscore character.

TOECatalog.TableNames

property TableNames: TStrings;

property TStrings* TableNames;

Description

This property returns the list of table names at the DataSource as identified by the Hdbc property.

TOECatalog.TableOwner

property TableOwner: **String**;

property AnsiString TableOwner;

Description

This property contains a mask used to limit the tables retrieved from the DataSource. The default value is an empty string, which indicates the specified tables of all users must be retrieved. The mask can contain the following meta-characters:

"_"	Underscore meta-character represents any single character.
"%"	Percent meta-character represents any sequence of zero or more characters.
"\"	Escape meta-character which permits the underscore, percent and escape meta-characters to be used as literal characters in the mask.

Example:

B_	Return all tables with owners that start with a "B" and are two characters long.
B%	Return all tables with owners that start with a B.
%_%	Return all tables with owners that contain an underscore character.

TOECatalog.Tables

property Tables: TCatalogTables;

property TCatalogTables* Tables;

Description

This property returns the list of tables at the DataSource as identified by the Hdbc property as TCatalogTable objects.

TOECatalog.TableType

TTableType = (ttTable, ttView, ttSystem);
TTableTypeSet = **set of** TTableType;

property TableType: TTableTypeSet;

TTableType {ttTable, ttView, ttSystem};
Set(TTableType, ttTable, ttSystem) TTableTypeSet;

property TableType: TTableTypeSet;

Description

This property indicates the type of tables to retrieve from the DataSource. This is a set property, which can contain any combination of the TTableType constants ttTable, ttView and ttSystem, which respectively indicates Tables, Views and System Tables. For some databases this property has to be an empty set to allow the tables to be retrieved from the database, however if a TTableType constant which isn't supported by the ODBC Driver is included in the set, it will be ignored. The default value is [ttTable, ttView].

TOECatalog.Terminate

procedure Terminate;

void Terminate(**void**);

Description

Used to close the cursors of result sets used within the component if necessary. This is to avoid "cursor open" errors with some ODBC drivers.

TCatalogTables

Hierarchy Properties

Unit OVCL;

Description

Maintains the properties of all the tables in a TOECatalog component.

TCatalogTables.ItemCount

property ItemCount: **Integer**;

property int ItemCount;

Description

Specifies the number of tables in the component.

TCatalogTables.Items

property Items[Index: **Integer**]: TCatalogTable;

property TCatalogTable* Items[**int** Index];

Description

Used to access the table at position Index in the component. Because this is the default property, Tables.Items[Index] is equivalent to Tables[Index].

TCatalogTable

Hierarchy Properties Methods

Unit OVCL;

Description

Maintains the properties of a table in a TOECatalog component.

TCatalogTable.ColumnByName

function ColumnByName(AColumnName: **String**): TCatalogColumn;

TCatalogColumn* ColumnByName(**AnsiString** AColumnName);

Description

This method returns the column object TCatalogColumn which corresponds to the column name, as retrieved using the Columns and ColumnNames properties.

TCatalogTable.ColumnNames

property ColumnNames: TStrings;

property TStrings* ColumnNames;

Description

This property returns the list of column names of the table.

TCatalogTable.Columns

property Columns: TCatalogColumns;

property TCatalogColumns* Columns;

Description

This property returns the list of columns of the table as TCatalogColumn objects.

TCatalogTable.Description

property Description: **String**;

property AnsiString Description;

Description

Returns a description of the table.

TCatalogTable.ForeignKeys

property ForeignKeys: TStrings;

property TStrings* ForeignKeys;

Description

This property returns the list of columns in the table that refer to primary keys in other tables. Each string is of the following format:

<COLUMN NAME>;[<FOREIGN OWNER>.]<FOREIGN TABLE>;<FOREIGN COLUMN>

where

<COLUMN NAME> indicates a column of the current table

[<FOREIGN OWNER>.]<FOREIGN TABLE> indicates the foreign table, with an optional owner field

<FOREIGN COLUMN> indicates the foreign column which <COLUMN NAME> points to

This string can be parsed by the ParseForeignKey method.

TCatalogTable.ForeignReferences

property ForeignReferences: TStrings;

property TStrings* ForeignReferences;

Description

Returns the inverse foreign keys than returned by the existing ForeignKeys property. While the ForeignKeys property returns a list of foreign keys in the specified table, the ForeignReferences property returns the foreign keys in other tables that refer to the primary key in the specified table. Each string is of the following format:

<COLUMN NAME>;[<FOREIGN OWNER>.]<FOREIGN TABLE>;<FOREIGN COLUMN>

where

<COLUMN NAME> indicates a primary key column of the current table

[<FOREIGN OWNER>.]<FOREIGN TABLE> indicates the foreign table, with an optional owner field

<FOREIGN COLUMN> indicates the foreign column pointing to <COLUMN NAME>

This string can be parsed by the ParseForeignKey method.

TCatalogTable.Indexes

property Indexes: TStrings;

property TStrings* Indexes;

Description

This property returns the list of indexes associated with the table. Each string is of the following format:

<INDEX NAME>;U|N;<COLUMN NAME>[,<COLUMN NAME>]...

where

<INDEX NAME> indicates the name of the index

U|N indicates whether the index is unique (U) or not unique (N)

<COLUMN NAME>[,<COLUMN NAME>]... indicates the list of columns of the current table which make up the index

This string can be parsed by the ParseIndex method.

TCatalogTable.PrimaryKeys

property PrimaryKeys: TStrings;

property TStrings* PrimaryKeys;

Description

This property returns the list of columns that makes up the primary key of the table.

TCatalogTable.Refresh

procedure Refresh;

void Refresh(**void**);

Description

This method is used to refresh the values contained by the Columns and ColumnNames properties.

TCatalogTable.TableName

property TableName: **String**;

property AnsiString TableName;

Description

Returns the name of the table.

TCatalogTable.TableOwner

property TableOwner: **String**;

property AnsiString TableOwner;

Description

Returns the owner of the table.

TCatalogTable.TableType

TTableType = (ttTable, ttView, ttSystem);

property TableType: TTableType;

TTableType {ttTable, ttView, ttSystem};

property TTableType TableType;

Description

Indicates if the table is a normal table, a view or a system table.

TCatalogTable.UniqueKey

property UniqueKey: **String**;

property AnsiString UniqueKey;

Description

Returns the optimal set of columns that uniquely identifies a row in the table and is therefore an alternative to the PrimaryKeys property. The string is of the following format, indicating the list of columns of the current table which uniquely identifies a row:

<COLUMN NAME>[,<COLUMN NAME>]...

This string can be parsed by the ParseUniqueKey method.

TCatalogProcedures

Hierarchy		Properties

Unit OVCL;

Description

Maintains the properties of all the stored procedures in a TOECatalog component.

TCatalogProcedures.ItemCount

property ItemCount: **Integer**;

property int ItemCount;

Description

Specifies the number of procedures in the component.

TCatalogProcedures.Items

property Items[Index: **Integer**]: TCatalogProcedure;

property TCatalogProcedure* Items[**int** Index];

Description

Used to access the procedure at position Index in the component. Because this is the default property, Procedures.Items[Index] is equivalent to Procedures[Index].

TCatalogProcedure

Hierarchy Properties Methods

Unit OVCL;

Description

Maintains the properties of a stored procedure in a TOECatalog component.

TCatalogProcedure.ColumnByName

function ColumnByName(AColumnName: **String**): TCatalogColumn;

TCatalogColumn* ColumnByName(**AnsiString** AColumnName);

Description

This method returns the column object TCatalogColumn which corresponds to the column name, as retrieved using the Columns and ColumnNames properties.

TCatalogProcedure.ColumnNames

property ColumnNames: TStrings;

property TStrings* ColumnNames;

Description

This property returns the list of parameter and column names of the procedure.

TCatalogProcedure.Columns

property Columns: TCatalogColumns;

property TCatalogColumns* Columns;

Description

This property returns the list of parameters and columns of the procedure as TCatalogColumn objects.

TCatalogProcedure.ProcedureName

property ProcedureName: **String**;

property AnsiString ProcedureName;

Description

Returns the name of the procedure.

TCatalogProcedure.ProcedureOwner

property ProcedureOwner: **String**;

property AnsiString ProcedureOwner;

Description

Returns the owner of the procedure.

TCatalogProcedure.Description

property Description: **String**;

property AnsiString Description;

Description

Returns a description of the procedure.

TCatalogProcedure.ProcedureType

property ProcedureType: SQLSMALLINT;

property SQLSMALLINT ProcedureType;

Description

Indicates the return type of the procedure. It can be one of the following values:

SQL_PT_UNKNOWN	It cannot be determined whether the procedure returns a value or not.
SQL_PT_PROCEDURE	The returned object is a procedure, since it does not have a return value.
SQL_PT_FUNCTION	The returned object is a function, since it has a return value.

TCatalogProcedure.Refresh

procedure Refresh;

void Refresh(**void**);

Description

This method is used to refresh the values contained by the Columns and ColumnNames properties.

TCatalogColumns

Hierarchy Properties

Unit OVCL;

Description

Maintains the properties of all the columns of a table or stored procedure in a TOECatalog component.

TCatalogColumns.ItemCount

property ItemCount: **Integer**;

property int ItemCount;

Description

Specifies the number of columns in the table.

TCatalogColumns.Items

property Items[Index: **Integer**]: TCatalogColumn;

property TCatalogColumn* Items[**int** Index];

Description

Used to access the column at position Index in the table. Because this is the default property, Columns.Items[Index] is equivalent to Columns[Index].

TCatalogColumn

Hierarchy Properties

Unit OVCL;

Description

Maintains the properties of a column of a table or stored procedure in a TOECatalog component.

TCatalogColumn.ColumnType

property ColumnType: SQLSMALLINT;

property SQLSMALLINT ColumnType;

Description

Indicates the type of the column. It can be one of the following values:

SQL_RESULT_COL	A result set column.
SQL_RETURN_VALUE	The return value of a procedure.
SQL_PARAM_INPUT	An input parameter of a procedure.
SQL_PARAM_OUTPUT	An output parameter of a procedure.
SQL_PARAM_INPUT_OUTPUT	An input/output parameter of a procedure.
SQL_PARAM_TYPE_UNKNOWN	A parameter of a procedure of which it is not possible to determine if it is an input, output or input/output parameter.

TCatalogColumn.DataType

property DataType: SQLSMALLINT;

property SQLSMALLINT DataType;

Description

Returns the database data type of a column.

TCatalogColumn.Default

property Default: **String**;

property AnsiString Default;

Description

Specifies the default value of the column, if any. For character data, quotes are included.

Example:

The string 'MALE' can be a default value.
The number 0 can be a default value.
The function {fn NOW()} can be a default value.

TCatalogColumn.ColumnName

property ColumnName: **String**;

property AnsiString ColumnName;

Description

Returns the name of the column.

TCatalogColumn.Nullable

property Nullable: SQLSMALLINT;

property SQLSMALLINT Nullable;

Description

Indicates if the column accepts NULL values or not. It can be one of the following values:

SQL_NO_NULLS	The column does not accepts NULL values.
SQL_NULLABLE	The column accepts NULL values.
SQL_NULLABLE_UNKNOWN	It is not known if the column accepts NULL values or not.

TCatalogColumn.Precision

property Precision: SQLINTEGER;

property SQLINTEGER Precision;

Description

Returns the precision of the column, or 0 if precision is not applicable. The precision of a numeric column refers to the maximum number of digits used by the data type of the column. The precision of a non-numeric column refers to the maximum length of the column.

TCatalogColumn.Radix

property Radix: SQLSMALLINT;

property SQLSMALLINT Radix;

Description

Returns the Radix of the column. For a numeric column, the radix is either 10 or 2. If 10, then Precision and Scale indicates the number of decimal digits allowed for the numeric column. If 2 then Precision and Scale indicates the number of bits allowed for the numeric column.

TCatalogColumn.Description

property Description: **String**;

property AnsiString Description;

Description

Returns a description of the column.

TCatalogColumn.Scale

property Scale: SQLSMALLINT;

property SQLSMALLINT Scale;

Description

Returns the scale of the column, or 0 if scale is not applicable. The scale of a numeric column refers to the maximum number of digits to the right of the decimal point.

TCatalogColumn.DataTypeName

property DataTypeName: **String**;

property AnsiString DataTypeName;

Description

Returns the DataSource-dependant database data type name of a column.

TOESchema

Hierarchy Properties Methods Events

Unit OVCL;

Description

The TOESchema Non-Visual Component is used to add and remove tables, indexes and views using generic SQL.

TOESchema.Abort

procedure Abort;

void Abort(**void**);

Description

Used to abort the loading or dropping of schema information. Application messages will be processed during these operations.

TOESchema.Aborted

property Aborted: **Boolean**;

property bool Aborted;

Description

Used to determine if the loading or dropping of schema information was aborted using the Abort method.

TOESchema.Clear

procedure Clear;

void Clear(**void**);

Description

Used to clear the tables and views from the component.

TOESchema.DoProgress

procedure DoProgress(Info: **String**); **virtual**;

virtual void DoProgress(**AnsiString** Info);

Description

Calls the OnProgress event.

TOESchema.DropTables

procedure DropTables(IgnoreErrors: **Boolean**);

void DropTables(**bool** IgnoreErrors);

Description

Used to drop all the tables and indexes defined in the component from the DataSource identified by the Hdbc property. The tables and indexes will be removed in the reverse order that they are defined within the component. To ignore cases where a table does not exist, set IgnoreErrors to True.

TOESchema.DropViews

procedure DropViews(IgnoreErrors: **Boolean**);

void DropViews(**bool** IgnoreErrors);

Description

Used to drop all the views defined in the component from the DataSource identified by the Hdbc property. The views will be removed in the reverse order that they are defined within the component. To ignore cases where a view does not exist, set IgnoreErrors to True.

TOESchema.ExecMarker

property ExecMarker: **String**;

property AnsiString ExecMarker;

Description

Indicates the SQL statement separator used when generating or running a SQL script. The default value is the string 'go'. Each marker in the script must be the only string on a new line indicating the end of a SQL statement in the stript.

Example:

```
CREATE TABLE dbo.Courses (CCode char(10), CDescr char(20), CNumStuds smallint,
PRIMARY KEY (CCode))
go
CREATE UNIQUE INDEX CIndex1 ON dbo.Courses (CCode)
go
```

TOESchema.GenScript

procedure GenScript(FileName: **String**);

void GenScript(**AnsiString** FileName);

Description

Used to write the table, index and view creation SQL, as generated for the DataSource identified by the Hdbc property, to the script FileName. The data types written to the file is DataSource specific. The ExecMarker is written on a new line after the end of each SQL statement written to the file.

TOESchema.Hdbc

property hDbc: THdbc;

property THdbc* hDbc;

Description

Specifies the connection component which identifies the database in which the schema must be maintained. The default value is nil.

TOESchema.LoadTables

procedure LoadTables;

void LoadTables(**void**);

Description

Used to load all the tables and indexes defined in the component at the DataSource identified by the Hdbc property. The tables and indexes will be created in the order that they are defined within the component.

TOESchema.LoadViews

procedure LoadViews;

void LoadViews(**void**);

Description

Used to load all the views defined in the component at the DataSource identified by the Hdbc property. The views will be created in the order that they are defined within the component.

TOESchema.NameConstraints

property NameConstraints: **Boolean**;

property bool NameConstraints;

Description

Used to enable alternative SQL generation for table constraints set up in the component. The default value is False. When set to True, the generation of "CREATE TABLE" SQL will be compatible with a wider range of databases, for example file-based databases such as MS Access, dBase, FoxPro and Paradox. This includes primary key, foreign key, nullability and uniqueness constraints.

TOESchema.OnProgress

TStepEvent = **procedure** (Sender: TObject;
Info: **String**) **of object**;

property OnProgress: TStepEvent;

void (**closure*** TStepEvent)(TObject* Sender,
AnsiString Info);

property TStepEvent OnProgress;

Description

This event tracks the progress when loading or dropping schema information. The Info parameter passes the table or view (with the owner if any) which was loaded or dropped before the event was called.

TOESchema.RunScript

procedure RunScript(FileName: **String**);

void RunScript(**AnsiString** FileName);

Description

Used to run the SQL script FileName against the DataSource identified by the Hdbc property. The SQL statements in the file can be DataSource specific.

TOESchema.Tables

property Tables: TSchemaTables;

property TSchemaTables* Tables;

Description

Maintains the properties of all the tables defined for a TOESchema component.

A property editor exists for this property. It can be accessed via the Object Inspector.

TOESchema.Terminate

procedure Terminate;

void Terminate(**void**);

Description

Used to close the cursors of result sets used within the component if necessary. This is to avoid "cursor open" errors with some ODBC drivers.

TOESchema.Views

property Views: TSchemaViews;

property TSchemaViews* Views;

Description

Maintains the properties of all the views defined for a TOESchema component.

A property editor exists for this property. It can be accessed via the Object Inspector.

TSchemaTables

Hierarchy Properties Methods

Unit OVCL;

Description

Maintains the properties of all the tables defined for a TOESChema component.

TSchemaTables.AddItem

procedure AddItem;

void AddItem(**void**);

Description

Used to add a TSchemaTable object to the component.

TSchemaTables.Clear

procedure Clear;

void Clear(**void**);

Description

Used to clear the tables from the component.

TSchemaTables.DeleteItem

procedure DeleteItem(Index: **Integer**);

void DeleteItem(**int** Index);

Description

Used to delete the TSchemaTable object at position Index from the component.

TSchemaTables.ExchangeItems

procedure ExchangeItems(Index1, Index2: **Integer**);

void ExchangeItems(**int** Index1,
int Index2);

Description

Used to exchange the positions of the two TSchemaTable objects at positions Index1 and Index2 in the component.

TSchemaTables.InsertItem

procedure InsertItem(Index: **Integer**);

void InsertItem(**int** Index);

Description

Used to insert a TSchemaTable object into the component at position Index.

TSchemaTables.ItemCount

property ItemCount: **Integer**;

property int ItemCount;

Description

Specifies the number of tables defined in the component.

TSchemaTables.Items

property Items[Index: **Integer**]: TSchemaTable;

property TSchemaTable* Items[**int** Index];

Description

Used to access the table at position Index in the component. Because this is the default property, Tables.Items[Index] is equivalent to Tables[Index].

TSchemaTables.MoveItem

procedure MoveItem(Index, NewIndex: **Integer**);

```
void MoveItem(int Index,
              int NewIndex);
```

Description

Used to move the TSchemaTable object at position Index in the component to position NewIndex.

TSchemaTable

Hierarchy Properties Methods

Unit OVCL;

Description

Maintains the properties of a table defined in a TOESchema component.

TSchemaTable.Columns

property Columns: TSchemaColumns;

property TSchemaColumns* Columns;

Description

Maintains the columns of a table defined in a TOESchema component.

TSchemaTable.DropIndexes

procedure DropIndexes(IgnoreErrors: **Boolean**);

void DropIndexes(**bool** IgnoreErrors);

Description

Allows you to drop all the indexes associated with the current table. To ignore cases where an index does not exist, set IgnoreErrors to True.

TSchemaTable.DropTable

procedure DropTable;

void DropTable(**void**);

Description

Used to remove the current table at the DataSource identified by the Hdbc property.

TSchemaTable.Indexes

property Indexes: TSchemaIndexes;

property TSchemaIndexes* Indexes;

Description

Maintains the indexes on a table defined in a TOESchema component.

TSchemaTable.LoadIndexes

procedure LoadIndexes;

void LoadIndexes(**void**);

Description

Allows you to load all the indexes associated with the current table.

TSchemaTable.LoadTable

procedure LoadTable;

void LoadTable(**void**);

Description

Used to create the current table at the DataSource identified by the Hdbc property.

TSchemaTable.TableName

property TableName: **String**;

property AnsiString TableName;

Description

Specifies the name of the table.

TSchemaTable.TableOwner

property TableOwner: **String**;

property AnsiString TableOwner;

Description

Specifies the owner of the table. The default value is an empty string.

TSchemaColumns

Hierarchy Properties Methods

Unit OVCL;

Description

Maintains the columns of a table defined in a TOESchema component.

TSchemaColumns.AddItem

procedure AddItem;

void AddItem(**void**);

Description

Used to add a TSchemaColumn object to the table.

TSchemaColumns.Clear

procedure Clear;

void Clear(**void**);

Description

Used to clear the columns from the current table.

TSchemaColumns.DeleteItem

procedure DeleteItem(Index: **Integer**);

void DeleteItem(**int** Index);

Description

Used to delete the TSchemaColumn object at position Index from the table.

TSchemaColumns.ExchangeItems

procedure ExchangeItems(Index1, Index2: **Integer**);

void ExchangeItems(**int** Index1,
                    int Index2);

Description

Used to exchange the two TSchemaColumn objects at positions Index1 and Index2 in the table.

TSchemaColumns.InsertItem

procedure InsertItem(Index: **Integer**);

void InsertItem(**int** Index);

Description

Used to insert a TSchemaColumn object at position Index in the table.

TSchemaColumns.ItemCount

property ItemCount: **Integer**;

property int ItemCount;

Description

Specifies the number of columns defined for the table.

TSchemaColumns.Items

property Items[Index: **Integer**]: TSchemaColumn;

property TSchemaColumn* Items[**int** Index];

Description

Used to access column Index of the table. Because this is the default property, Columns.Items[Index] is equivalent to Columns[Index].

TSchemaColumns.MoveItem

procedure MoveItem(Index, NewIndex: **Integer**);

void MoveItem(**int** Index,
                    int NewIndex);

Description

Used to move the TSchemaColumn object at position Index in the table to position NewIndex.

TSchemaColumn

Hierarchy   Properties

Unit OVCL;

Description

Maintains the properties of a column in a table of a TOESchema component.

TSchemaColumn.Default

property Default: **String**;

property AnsiString Default;

Description

Specifies the default value of the current column, if any. For character data, quotes must be included.

Example:

The string 'MALE' can be a default value.
The number 0 can be a default value.
The function {fn NOW()} can be a default value.

TSchemaColumn.Flags

property Flags: **Integer**;

property int Flags;

Description

This property value can be any combination of the following flags:

cfNotNull	Specifies that the current column cannot contain a null value.
cfUnique	Specifies that each value in the current column is unique.
cfPrimaryKey	Specifies that the current column is part of the primary key of the current table.

TSchemaColumn.ColumnName

property ColumnName: **String**;

property AnsiString ColumnName;

Description

Specifies the name of the column.

TSchemaColumn.Precision

property Precision: **Integer**;

property int Precision;

Description

Specifies the size in bytes of the column.

TSchemaColumn.DataType

property DataType: SQLSMALLINT;

property SQLSMALLINT DataType;

Description

Specifies the database data type of the column.

TSchemaColumn.ForeignColumn

property ForeignColumn: **String**;

property AnsiString ForeignColumn;

Description

Indicates the foreign column in the foreign table ForeignTable of the column, if any.

TSchemaColumn.ForeignTable

property ForeignTable: **String**;

property AnsiString ForeignTable;

Description

Indicates the foreign table which contains the foreign column ForeignColumn of the column, if any.

TSchemaIndexes

Hierarchy Properties Methods

Unit OVCL;

Description

Maintains the indexes on a table defined in a TOESchema component.

TSchemaIndexes.AddItem

procedure AddItem;

void AddItem(**void**);

Description

Used to add a TSchemaIndex object to the table.

TSchemaIndexes.Clear

procedure Clear;

void Clear(**void**);

Description

Used to clear the indexes from the current table.

TSchemaIndexes.DeleteItem

procedure DeleteItem(Index: **Integer**);

void DeleteItem(**int** Index);

Description

Used to delete the TSchemaIndex object at position Index in the table.

TSchemaIndexes.ExchangeItems

procedure ExchangeItems(Index1, Index2: **Integer**);

void ExchangeItems(**int** Index1,
int Index2);

Description

Used to exchange the TSchemaIndex objects at positions Index1 and Index2 in the table.

TSchemaIndexes.InsertItem

procedure InsertItem(Index: **Integer**);

void InsertItem(**int** Index);

Description

Used to insert a TSchemaIndex object at position Index in the table.

TSchemaIndexes.ItemCount

property ItemCount: **Integer**;

property int ItemCount;

Description

Specifies the number of indexes defined for the table.

TSchemaIndexes.Items

property Items[Index: **Integer**]: TSchemaIndex;

property TSchemaIndex* Items[**int** Index];

Description

Used to access the Indexes defined for the table. Because this is the default property, Indexes.Items[Index] is equivalent to Indexes[Index].

TSchemaIndexes.MoveItem

procedure MoveItem(Index, NewIndex: **Integer**);

void MoveItem(**int** Index,
int NewIndex);

Description

Used to move the TSchemaIndex object at position Index in the table to position NewIndex.

TSchemaIndex

Hierarchy Properties Methods

Unit OVCL;

Description

Maintains the properties of an index of a table defined in a TOESchema component.

TSchemaIndex.Columns

property Columns: TStrings;

property TStrings* Columns;

Description

Specifies the defined table columns on which the index is created.

TSchemaIndex.DropIndex

procedure DropIndex;

void DropIndex(**void**);

Description

Used to remove the current index from the DataSource identified by the Hdbc property.

TSchemaIndex.IndexName

property IndexName: **String**;

property AnsiString IndexName;

Description

Specifies the name of the index.

TSchemaIndex.LoadIndex

procedure LoadIndex;

void LoadIndex(**void**);

Description

Used to create the current index at the DataSource identified by the Hdbc property.

TSchemaIndex.Unique

property Unique: **Boolean**;

property bool Unique;

Description

Specifies whether the index is an unique index or not. Set to True to indicate an unique index, otherwise False.

TSchemaViews

Hierarchy Properties Methods

Unit OVCL;

Description

Maintains the properties of all the views defined for a TOESChema component.

TSchemaViews.AddItem

procedure AddItem;

void AddItem(**void**);

Description

Used to add a TSchemaView object to the component.

TSchemaViews.Clear

procedure Clear;

void Clear(**void**);

Description

Used to clear the views from the component.

TSchemaViews.DeleteItem

procedure DeleteItem(Index: **Integer**);

void DeleteItem(**int** Index);

Description

Used to delete the TSchemaView object at position Index from the component.

TSchemaViews.ExchangeItems

procedure ExchangeItems(Index1, Index2: **Integer**);

void ExchangeItems(**int** Index1,
int Index2);

Description

Used to exchange the two TSchemaView objects at positions Index1 and Index2 in the component.

TSchemaViews.InsertItem

procedure InsertItem(Index: **Integer**);

void InsertItem(**int** Index);

Description

Used to insert a TSchemaView object at position Index in the component.

TSchemaViews.ItemCount

property ItemCount: **Integer**;

property int ItemCount;

Description

Specifies the number of views defined in the component.

TSchemaViews.Items

property Items[Index: **Integer**]: TSchemaView;

property TSchemaView* Items[**int** Index];

Description

Used to access the views defined in the component. Because this is the default property, Views.Items[Index] is equivalent to Views[Index].

TSchemaViews.MoveItem

procedure MoveItem(Index, NewIndex: **Integer**);

void MoveItem(**int** Index,

int NewIndex);

Description

Used to move the TSchemaView object at position Index in the component to position NewIndex.

TSchemaView

Hierarchy Properties Methods

Unit OVCL;

Description

Maintains the properties of a view of a table defined in a TOESchema component.

TSchemaView.Columns

property Columns: TStrings;

property TStrings* Columns;

Description

Specifies the column names of the view given to the columns selected by the SelectSQL statement. If this string list is empty, the default column names as specified in the SelectSQL statement will be used as the column names of the view.

TSchemaView.DropView

procedure DropView;

void DropView(**void**);

Description

Used to remove the current view from the DataSource identified by the Hdbc property.

TSchemaView.LoadView

procedure LoadView;

void LoadView(**void**);

Description

Used to create the current view at the DataSource identified by the Hdbc property.

TSchemaView.SelectSQL

property SelectSQL: **String**;

property AnsiString SelectSQL;

Description

Specifies the select SQL statement used to construct the view.

TSchemaView.ViewName

property ViewName: **String**;

property AnsiString ViewName;

Description

Specifies the name of the view.

TSchemaView.ViewOwner

property ViewOwner: **String**;

property AnsiString ViewOwner;

Description

Specifies the owner of the view. The default value is an empty string.

TOEBulkCopy

Hierarchy	Properties	Methods	Events

Unit OVCL;

Description

The TOEBulkCopy Non-Visual Component is used to easily copy data from one data source to another data source.

TOEBulkCopy.Abort

procedure Abort;

void Abort(**void**);

Description

Used to abort the copying of data. Application messages will be processed during these operations.

TOEBulkCopy.Aborted

property Aborted: **Boolean**;

property bool Aborted;

Description

Used to determine if the copying of data was aborted using the Abort method.

TOEBulkCopy.CommitCount

property CommitCount: SQLUINTEGER;

property SQLUINTEGER CommitCount;

Description

Used to indicate how the copy process must be transactioned. It can be one of the following values:

0	Transactioning will be done, with a single commit after all the records where copied.
1	No transactioning will be done (auto-commit after each row inserted).
n > 1	Transactioning will be done, with a commit after every n rows copied.

The default value is 1.

TOEBulkCopy.DoBindParams

procedure DoBindParams(Table: **String**;
Hstmt: THstmt); **virtual**;

virtual void DoBindParams(**AnsiString** Table;
THstmt* Hstmt);

Description

Calls the OnBindParams event.

TOEBulkCopy.DoProgress

procedure DoProgress(Info: **String**); **virtual**;

virtual void DoProgress(**AnsiString** Info);

Description

Calls the OnProgress event.

TOEBulkCopy Events

TOEBulkCopy

published
OnProgress
OnBindParams

TOEBulkCopy.ExecMarker

property ExecMarker: **String**;

property AnsiString ExecMarker;

Description

Indicates the SQL statement separator used when running a SQL script. The default value is the string 'go'. Each marker in the script must be the only string on a new line indicating the end of a SQL statement in the stript.

Example:

```
SELECT CCode, CDescr, CNumStuds FROM Courses
go
INSERT INTO Courses (CCode, CDescr, CNumStuds) VALUES (?, ?, ?)
go
```

TOEBulkCopy.Execute

procedure Execute;

void Execute(**void**);

Description

Used to copy data from the source database to the target database. It makes use of SQLSource to retrieve the data from the source, and SQLTarget to insert the data at the target.

TOEBulkCopy.hDbcSource

property hDbcSource: THdbc;

property THdbc* hDbcSource;

Description

This property is set to the connection component which identifies the source database to copy data from.

TOEBulkCopy.hDbcTarget

property hDbcTarget: THdbc;

property THdbc* hDbcTarget;

Description

This property is set to the connection component which identifies the target database to copy data to.

Hierarchy

TComponent
  |
TOEBulkCopy

TOEBulkCopy.MaxRows

property MaxRows: SQLUINTEGER;

property SQLUINTEGER MaxRows;

Description

Allows you to limit the number of rows to be copied from the source database to the target database. The default value is 0, which indicates that the number of rows to copy is not limited. It is similar to the THstmt.MaxRows property.

TOEBulkCopy Methods

TOEBulkCopy

protected
  DoProgress
  DoBindParams

public
  Execute
  RunScript
  Terminate
  Abort

TOEBulkCopy.OnBindParams

TBindEvent = **procedure** (Sender: TObject;
Table: **String**;
Hstmt: THstmt) **of object**;

property OnBindParams: TBindEvent;

void (**closure*** TBindEvent)(Sender: TObject;
AnsiString Table;

```
                              THstmt* Hstmt);
```

property TBindEvent OnBindParams;

Description

Used to limit the rows copied from the source database by binding the parameters for the statement used to retrieve the data. The table parameter is used to identify the parameters to bound. Parameters can be defined for the SQLSource property used by the Execute method and for the source statements in the script used by the RunScript method.

TOEBulkCopy.OnProgress

```
TStepEvent = procedure (Sender: TObject;
                             Info: String) of object;
```

property OnProgress: TStepEvent;

```
void (closure* TStepEvent)(TObject* Sender,
                              AnsiString Info);
```

property TStepEvent OnProgress;

Description

This event tracks the progress when copying data. The Info parameter passes the table (with the owner if any) which is being copied when the event was called.

TOEBulkCopy Properties

TOEBulkCopy

public

- RowsAffected
- Aborted

published

- hDbcSource
- hDbcTarget
- SQLSource
- SQLTarget
- ExecMarker
- RowSetSize
- MaxRows
- CommitCount

TOEBulkCopy.RowsAffected

property RowsAffected: SQLINTEGER;

property SQLINTEGER RowsAffected;

Description

Returns the number of rows copied sofar from the source database to the target database.

TOEBulkCopy.RowSetSize

property RowSetSize: SQLUINTEGER;

property SQLUINTEGER RowSetSize;

Description

Used to set the number of rows that must be copied at a time from the source database to the target database. The default value is 1. It is similar to the THstmt.RowSetSize property.

TOEBulkCopy.RunScript

procedure RunScript(FileName: **String**);

void RunScript(**AnsiString** FileName);

Description

Used to copy data by running the SQL script FileName.

Important

The script must contain pairs of statements. The first statement in each pair must be similar as defined for SQLSource, and the second statement in each pair must be similar as defined for SQLTarget. Each statement must end with the ExecMarker.

Example:

```
SELECT SNo, SName, SBirth FROM Students
go
INSERT INTO Students (SNo, SName, SBirth) VALUES (?, ?, ?)
go
SELECT CCode, CDescr, CNumStuds FROM Courses
go
INSERT INTO Courses (CCode, CDescr, CNumStuds) VALUES (?, ?, ?)
go
...
```

TOEBulkCopy.SQLSource

property SQLSource: **String**;

property AnsiString SQLSource;

Description

This property is set to a SQL statement which generates a result set, normally a select statement. It is used to copy data from the source database using the Execute method.

TOEBulkCopy.SQLTarget

property SQLTarget: **String**;

property AnsiString SQLTarget;

Description

This property is set to a SQL statement with insert parameters, normally an insert statement. It is used to copy data to the target database using the Execute method..

TOEBulkCopy.Terminate

procedure Terminate;

void Terminate(**void**);

Description

Used to close the cursors of result sets used within the component if necessary. This is to avoid "cursor open" errors with some ODBC drivers.

TOEAdministrator

Hierarchy Properties Methods

Unit OVCL;

Description

The TOEAdministrator Non-Visual Component maintains ODBC DataSources for an application. It is used to add, modify, remove and list ODBC DataSources.

TOEAdministrator.Add

function Add: **Boolean**;

bool Add(**void**);

Description

Used to create a new DataSource, using the current properties of the component. The Driver property is compulsory for this method. If the DataSourceType is dsSystem the DataSource will be added as a system DataSource, otherwise it will be added as a user DataSource.

TOEAdministrator.Attributes

property Attributes: TStrings;

property TStrings* Attributes;

Description

Specifies additional ODBC Driver-dependant attributes used when adding or modifying a DataSource. Each string of the Attributes property must be a keyword-value pair, using the syntax <KEYWORD>=<VALUE>.

Example:

Delphi Code

```
with OEAdministrator1 do
begin
  //set new DataSource and ODBC Driver
  DataSource:= 'dsnStudents';
  Driver:= 'SQL Server';

  //add Server and Database attributes
  Attributes.Clear;
  Attributes.Add('SERVER=JUBILEE');
  Attributes.Add('DATABASE=STUDENTSDB');

  //create the DataSource
  Add;
end;
```

C++ Code

```
//set new DataSource and ODBC Driver
OEAdministrator1->DataSource = "dsnStudents";
OEAdministrator1->Driver = "SQL Server";

//add Server and Database attributes
OEAdministrator1->Attributes->Clear();
OEAdministrator1->Attributes->Add("SERVER=JUBILEE");
OEAdministrator1->Attributes->Add("DATABASE=STUDENTSDB");

//create the DataSource
OEAdministrator1->Add();
```

TOEAdministrator.DataSource

property DataSource: **String**;

property AnsiString DataSource;

Description

Specifies the DataSource name to be added, modified or deleted. Before setting this property in code, use the Valid method to check if the DataSource name doesn't contain any illegal characters.

TOEAdministrator.DataSourceDriver

function DataSourceDriver(ADataSource: **String**): **String**;

AnsiString DataSourceDriver(**AnsiString** ADataSource);

Description

This method returns the ODBC Driver name associated with the given existing DataSource, as returned by the DataSources property.

TOEAdministrator.DataSources

property DataSources: TStrings;

property TStrings* DataSources;

Description

This property returns the list of ODBC DataSources which exist on the local machine.

TOEAdministrator.DataSourceType

TDataSourceType = (dsDefault, dsUser, dsSystem);

property DataSourceType: TDataSourceType;

TDataSourceType {dsDefault, dsUser, dsSystem};

property TDataSourceType DataSourceType;

Description

Specifies the type of DataSources to return for the DataSources property, according to the following table:

dsDefault	User and System DataSources.
dsUser	User DataSources.
dsSystem	System DataSources.

It is also used to specify whether to create a user DataSource (dsDefault or dsUser) or system DataSource (dsSystem) when adding a DataSource. The default value is dsDefault. A System DataSource is usually required for use by system processes.

TOEAdministrator.Driver

property Driver: **String**;

property AnsiString Driver;

Description

Specifies the ODBC Driver used when creating or modifying a DataSource. This property is compulsory in DataSource creation.

TOEAdministrator.Drivers

property Drivers: TStrings;

property TStrings* Drivers;

Description

This property returns the list of ODBC Drivers installed on the local machine.

TOEAdministrator.Modify

function Modify: **Boolean**;

bool Modify(**void**);

Description

Used to modify the attributes of a DataSource. The DataSource and Driver properties are compulsory for this method. If the DataSourceType is dsSystem the system DataSource will be modified, otherwise the user DataSource will be modified.

TOEAdministrator.Password

property Password: **String**;

property AnsiString Password;

Description

Specifies the account access code used when creating or modifying a DataSource. Usually DataSources aren't created or modified to hold account information, but rather account information is supplied during connection to the database.

TOEAdministrator.Prompt

property Prompt: **Boolean**;

property bool Prompt;

Description

Specifies whether to prompt the user to complete information when adding, modifying or deleting DataSources. Set to True to allow user prompting, otherwise False. The default value is False.

TOEAdministrator.Refresh

procedure Refresh;

void Refresh(**void**);

Description

This method is used to refresh the string lists returned by the DataSources and Drivers properties.

TOEAdministrator.Remove

function Remove: **Boolean**;

bool Remove(**void**);

Description

Used to remove a DataSource. The DataSource and Driver properties are compulsory for this method. If the DataSourceType is dsSystem the system DataSource will be removed, if the DataSourceType is dsUser the user DataSource will be removed, otherwise the default DataSource will be removed.

TOEAdministrator.UserName

property UserName: **String**;

property AnsiString UserName;

Description

Specifies the account used when creating or modifying a DataSource. Usually DataSources aren't created or modified to hold account information, but rather account information is supplied during connection to the database.

TOEAdministrator.Valid

function Valid(ADataSource: **String**): **Boolean**;

bool Valid(**AnsiString** ADataSource);

Description

This method is used to determine if the string passed can be a valid DataSource name, by checking that it does not contain any invalid characters. It returns True if the string can be a valid DataSource name. This method is used before setting the DataSource property in code.

TOESetup

Hierarchy Properties Methods Events Example

Unit OSI;

Description

The TOESetup component is used to install and configure ODBC, ODBC Drivers and ODBC Translators on a target machine. The necessary configuration will be done by the component, while three methods of actual file copying are provided:

Default copy procedures, which will even defer copying if files are in use.
Events called for each file to be copied can be used to copy the files.
For Win95 and WinNT a system driven INF file copy can be done.

The configuration process is driven by properties of the component, while the copying process is completely driven by an INF file. The INF file is also written in such a way that a right-click-install can be done on the INF file from Explorer to copy the files to the correct locations.

TOESetup.Driver

property Driver: TDriverSetup;

property TDriverSetup* Driver;

Description

The TDriverSetup class is used to install and configure an ODBC Driver on a target machine.

TOESetup.InfCopy

property InfCopy: **Boolean**;

property bool InfCopy;

Description

Specifies whether a system INF file copy must be done, or whether a component or event driven copy must be done to place the files on the target machine. The default value is False. It is advisable to reboot the machine after an INF file copy was done, since some existing files on the target machine might have been in use during the copy.

Note:

This property is not supported under 16bit Windows, since 16bit Windows doesn't support system INF file copying.

TOESetup.InfFile

property InfFile: **String**;

property AnsiString InfFile;

Description

Used to specify the INF file to be used during the ODBC and ODBC Driver installation. The default value is the string 'ODBC.INF'.

TOESetup.Install

function Install: **Boolean**;

bool Install(**void**);

Description

Used to install the sections in the INF file as specified by the CopyFiles keyword under the [DefaultInstall] section of the INF file. The CopyFiles keyword can list any combination of the [Manager], [Driver] and [Translator] sections in the INF file.

Example:

To install the [Manager] and [Driver] sections in the INF file, the [DefaultInstall] section in the INF file must look as follows:

```
[DefaultInstall]
CopyFiles=Manager, Driver
```

TOESetup.Manager

property Manager: TManagerSetup;

property TManagerSetup* Manager;

Description

The TManagerSetup class is used to install and configure ODBC on a target machine.

TOESetup.OnCopyFile

```
TCopyFileEvent = function (Sender: TObject;
                          FromFile: String;
                          ToFile: String): Boolean of object;
```

property OnCopyFile: TCopyFileEvent;

```
bool (closure* TCopyFileEvent)(TObject* Sender,
                              AnsiString FromFile,
                              AnsiString ToFile);
```

property TCopyFileEvent OnCopyFile;

Description

This event is called for each file that needs to be copied. FromFile specifies the full path and filename of the file to be copied and ToFile specifies the full path and filename of the destination file. Return False from the event when custom copying of the file is done, otherwise return True from the event to allow copying of the file by the TOESetup component. This event can also be used to keep track of the progress of the copy.

TOESetup.OverWrite

TOverWrite = (owAlways, owNever, owOlder);

property OverWrite: TOverWrite;

TOverWrite {owAlways, owNever, owOlder};

property TOverWrite OverWrite;

Description

Used to specify what action to take when a file to be copied already exists on the target machine. The default value is owAlways. It can be one of the following values:

owAlways	Always overwrite the existing file on the machine with the file being copied.
owNever	Always skip copying a file when there's an existing file on the machine.
owOlder	Only overwrite the existing file if it is older than the file being copied.

If an existing file is in use and the component is instructed to overwrite the file, a deferred copy of the file will be done to the target machine. These files will automatically be placed in the correct directories on the next reboot of the target machine. It is therefore advisable to reboot the machine after a component copy was done.

TOESetup.Translator

property Translator: TTranslatorSetup;

property TTranslatorSetup* Translator;

Description

The TTranslatorSetup class is used to install and configure an ODBC Translator on a target machine.

TOESetup Example

To install the ODBC Driver Manager and the Microsoft SQL Server ODBC Driver using class copying, the following code is needed:

Delphi Code

```
begin
  //specify the driver description and DLL
  OESetup1.Driver.DriverDesc:= 'SQL Server';
  OESetup1.Driver.DriverDLL:= 'sqlsrv32.dll';
  //install the [DefaultInstall] section of the ODBC.INF file
  OESetup1.Install;
end;
```

C++ Code

```
//specify the driver description and DLL
OESetup1->Driver->DriverDesc = "SQL Server";
OESetup1->Driver->DriverDLL = "sqlsrv32.dll";

//install the [DefaultInstall] section of the ODBC.INF file
OESetup1->Install();
```

The InfFile property is left at its default value of ODBC.INF and the OverWrite property is left at its default value of owAlways to always overwrite existing files on the target machine. The ODBC.INF file looks as follows:

```
[Version]
;This section is needed to do a right-click-install on the INF file from Explorer
Signature=$Chicago$
Provider=ODBCExpress

[SourceDisksNames]
;This section specifies the source directory in which the files to be installed are located
1="ODBC",,,.\

[DestinationDirs]
;This section specifies the destination directory of the files to be installed
DefaultDestDir=11  ;System Directory

[DefaultInstall]
;This section specifies which sections of the INF file must be installed - only Manager and Driver
sections in this case
;CopyFiles=Manager, Driver, Translator  ;Commented Out
CopyFiles=Manager, Driver

[Manager]
;Administrator
odbcad32.ex_=odbcad32.exe

;Code Page
12520437.cp_=12520437.cpx
12520850.cp_=12520850.cpx
mscpxl32.dl_=mscpxl32.dll

;Connection Pooling
mtxdm.dl_=mtxdm.dll

;Driver Manager
ds16gt.dl_=ds16gt.dll
ds32gt.dl_=ds32gt.dll
odbc16gt.dl_=odbc16gt.dll
odbc32.dl_=odbc32.dll
odbc32gt.dl_=odbc32gt.dll
odbccp32.cp_=odbccp32.cpl
```

```
odbccp32.dll=odbccp32.dll
odbccr32.dl_=odbccr32.dll
odbcint.dll=odbcint.dll
odbctrac.dl_=odbctrac.dll

;Help
odbcinst.cn_=odbcinst.cnt
odbcinst.hl_=odbcinst.hlp

;Support Files
msvcrt40.dll=msvcrt40.dll

[Driver]
;SQL Server
sqlsrv32.dl_=sqlsrv32.dll  ;Driver DLL
                  ;Setup DLL
dbnmpntw.dl_=dbnmpntw.dll  ;Network
drvssrvr.hl_=drvssrvr.hlp  ;Help

[Translator]
```

TManagerSetup

Hierarchy   Methods

Unit OSI;

Description

The TManagerSetup class is used to install and configure ODBC on a target machine.

TManagerSetup.Install

function Install: **Boolean**;

bool Install(**void**);

Description

Installs only the [Manager] section of the INF file, irrespective of the sections to be installed listed under the [DefaultInstall] section.

TDriverSetup

Hierarchy   Properties   Methods

Unit OSI;

Description

The TDriverSetup class is used to install and configure an ODBC Driver on a target machine.

TDriverSetup.DriverDesc

property DriverDesc: **String**;

property AnsiString DriverDesc;

Description

Specifies the descriptive name of the ODBC Driver to install. This is the same name as displayed for the Driver in the ODBC Administrator, usually located in the Control Panel.

Example:

For the Microsoft SQL Server ODBC Driver, the descriptive name is 'SQL Server'.

TDriverSetup.DriverDLL

property DriverDLL: **String**;

property AnsiString DriverDLL;

Description

Specifies the name of the ODBC Driver DLL to install for the driver specified in DriverDesc.

Example:

The 32bit Microsoft SQL Server ODBC Driver is called 'sqlsrv32.dll'.

TDriverSetup.DriverSection

property DriverSection: **String**;

property AnsiString DriverSection;

Description

Specifies the name of the ODBC Driver section in the .INF file to install for the driver specified in DriverDesc.

TDriverSetup.Install

function Install: **Boolean**;

bool Install(**void**);

Description

Installs only the [Driver] section of the INF file, irrespective of the sections to be installed listed under the [DefaultInstall] section.

TDriverSetup.SetupDLL

property SetupDLL: **String**;

property AnsiString SetupDLL;

Description

Specifies the name of the ODBC Driver Setup DLL that must be used to install the driver specified in DriverDesc. In some cases the Driver DLL and Setup DLL is the same DLL, which means this property must be left at its default value, an empty string.

Example:

The 32bit Microsoft SQL Server Driver DLL and Setup DLL is the same DLL, 'sqlsrv32.dll'. Therefore this property must be set to an empty string when installing this ODBC Driver.

TTranslatorSetup

Hierarchy Properties Methods

Unit OSI;

Description

The TTranslatorSetup class is used to install and configure an ODBC Translator on a target machine.

TTranslatorSetup.Install

function Install: **Boolean**;

bool Install(**void**);

Description

Installs only the [Translator] section of the INF file, irrespective of the sections to be installed listed under the [DefaultInstall] section.

TTranslatorSetup.SetupDLL

property SetupDLL: **String**;

property AnsiString SetupDLL;

Description

Specifies the name of the ODBC Translator Setup DLL that must be used to install the translator specified in TranslatorDesc. In some cases the Translator DLL and Setup DLL is the same DLL, which means this property must be left at its default value, an empty string.

TTranslatorSetup.TranslatorDesc

property TranslatorDesc: **String**;

property AnsiString TranslatorDesc;

Description

Specifies the descriptive name of the ODBC Translator to install.

TTranslatorSetup.TranslatorDLL

property TranslatorDLL: **String**;

property AnsiString TranslatorDLL;

Description

Specifies the name of the ODBC Translator DLL to install for the translator specified in TranslatorDesc.

TTranslatorSetup.TranslatorSection

property TranslatorSection: **String**;

property AnsiString TranslatorSection;

Description

Specifies the name of the ODBC Translator section to install for the translator specified in TranslatorDesc.

TOEUpdateObject

Hierarchy Properties Methods

Unit ODSI;

Description

The TOEUpdateObject component is similar to the BDE TUpdateObject component and is the base class for the components used to perform positional insert, update and delete operations (level 4 SQL Generation).

TOEUpdateObject.Apply

procedure Apply(UpdateKind: TUpdateKind); **virtual**; **abstract**;

virtual void Apply(TUpdateKind UpdateKind) = 0;

Description

This method is used to perform the requested insert, update or delete operation of the UpdateObject component. It has to be implemented by descendant components.

TOEUpdateObject.DataSet

property DataSet: TCustomOEDataSet;

property TCustomOEDataSet* DataSet;

Description

Used to set the TCustomOEDataSet the UpdateObject applies to.

TOEUpdateObject.GetDataSet

function GetDataSet: TCustomOEDataSet; **virtual**; **abstract**;

virtual TCustomOEDataSet* GetDataSet(**void**) = 0;

Description

The 'get' method of the DataSet property has to be implemented by descendant components.

TOEUpdateObject.GetImplemented

function GetImplemented: **Boolean**; **virtual**; **abstract**;

virtual bool GetImplemented(**void**) = 0;

Description

The 'get' method of the Implemented property has to be implemented by descendant components.

TOEUpdateObject.Implemented

property Implemented[UpdateKind: TUpdateKind]: **Boolean**;

property bool Implemented[TUpdateKind UpdateKind];

Description

Used to determine whether the insert, update or delete operation of the UpdateObject is implemented. If a specific operation is not implemented, another level of SQL Generation will be used instead.

TOEUpdateObject.SetDataSet

procedure SetDataSet(ADataSet: TCustomOEDataSet); **virtual**; **abstract**;

virtual void SetDataSet(TCustomOEDataSet* ADataSet) = 0;

Description

The 'set' method of the DataSet property has to be implemented by descendant components.

TOEUpdateSQL

Hierarchy Properties Methods

Unit ODSI;

Description

The TOEUpdateSQL component is similar to the BDE TUpdateSQL component and is used to perform positional insert, update and delete operations (level 4 SQL Generation) using SQL statements.

TOEUpdateSQL.DeleteSQL

property DeleteSQL: TStrings;

property TStrings* DeleteSQL;

Description

Indicates the SQL statement used to perform the delete operations.

TOEUpdateSQL.ExecSQL

procedure ExecSQL(UpdateKind: TUpdateKind);

void ExecSQL(TUpdateKind UpdateKind);

Description

Executes the appropriate SQL statement to perform the specified operation.

TOEUpdateSQL.InsertSQL

property InsertSQL: TStrings;

property TStrings* InsertSQL;

Description

Indicates the SQL statement used to perform the insert operations.

TOEUpdateSQL.ModifySQL

property ModifySQL: TStrings;

property TStrings* ModifySQL;

Description

Indicates the SQL statement used to perform the update operations.

TOEUpdateSQL.Query

property Query[UpdateKind: TUpdateKind]: TCustomOEQuery;

property TCustomOEQuery* Query[TUpdateKind UpdateKind];

Description

Returns the TCustomOEQuery component used to perform the specified operation.

TOEUpdateSQL.SetParams

procedure SetParams(UpdateKind: TUpdateKind);

void SetParams(TUpdateKind UpdateKind);

Description

Assigns the parameters to the appropriate Query component prior to performing the specified operation.

TOEUpdateSQL.SQL

property SQL[UpdateKind: TUpdateKind]: TStrings;

property TStrings* SQL[TUpdateKind UpdateKind];

Description

Returns the SQL statement used to perform the specified operation.

TOEUpdateProc

Hierarchy Properties Methods

Unit ODSI;

Description

The TOEUpdateProc component is similar to the BDE TUpdateSQL component and is used to perform positional insert, update and delete operations (level 4 SQL Generation) using stored procedures. It also retrieves the output values of these stored procedures.

TOEUpdateProc.DeleteParams

property DeleteParams: TOEParams;

property TOEParams* DeleteParams;

Description

Indicates the parameters of the stored procedure used to perform the delete operations.

TOEUpdateProc.DeleteProc

property DeleteProc: **String**;

property AnsiString DeleteProc;

Description

Indicates the stored procedure used to perform the delete operations.

TOEUpdateProc.ExecSQL

procedure ExecSQL(UpdateKind: TUpdateKind);

void ExecSQL(TUpdateKind UpdateKind);

Description

Executes the appropriate stored procedure to perform the specified operation.

TOEUpdateProc.InsertParams

property InsertParams: TOEParams;

property TOEParams* InsertParams;

Description

Indicates the parameters of the stored procedure used to perform the insert operations.

TOEUpdateProc.InsertProc

property InsertProc: **String**;

property AnsiString InsertProc;

Description

Indicates the stored procedure used to perform the insert operations.

TOEUpdateProc.ModifyParams

property ModifyParams: TOEParams;

property TOEParams* ModifyParams;

Description

Indicates the parameters of the stored procedure used to perform the update operations.

TOEUpdateProc.ModifyProc

property ModifyProc: **String**;

property **AnsiString** ModifyProc;

Description

Indicates the stored procedure used to perform the update operations.

TOEUpdateProc.Params

property Params[UpdateKind: TUpdateKind]: TOEParams;

property TOEParams* Params[TUpdateKind UpdateKind];

Description

Returns the parameters of the stored procedure used to perform the specified operation.

TOEUpdateProc.Query

property Query[UpdateKind: TUpdateKind]: TCustomOEQuery;

property TCustomOEQuery* Query[TUpdateKind UpdateKind];

Description

Returns the TCustomOEQuery component used to perform the specified operation.

TOEUpdateProc.SetParams

procedure SetParams(UpdateKind: TUpdateKind);

void SetParams(TUpdateKind UpdateKind);

Description

Assigns the parameters to the appropriate Query component prior to performing the specified operation.

TOEUpdateProc.StoredProc

property StoredProc[UpdateKind: TUpdateKind]: **String**;

property AnsiString StoredProc[TUpdateKind UpdateKind];

Description

Returns the stored procedure used to perform the specified operation.

TOEIntFloat

Hierarchy Properties Methods

Unit OVCL;

Description

This data-aware Visual Control is used to maintain integer and float fields from a DataSource. It can handle one column of one row from the result set of a TDataSet at a time, hence it is called a single-row control.

TOEIntFloat.DataField

property DataField: **String**;

property AnsiString DataField;

Description

This is used to set the column in the result set that the control must be populated with.

TOEIntFloat.DataSource

property DataSource: TDataSource;

property TDataSource* DataSource;

Description

This is used to set the TDataSource component that links the control with a result set.

TOEIntFloat.Field

property Field: TField;

property TField* Field;

Description

This returns the column TField object that the control is associated with.

TOEIntFloat.ReadOnly

property ReadOnly: **Boolean**;

property bool ReadOnly;

Description

This is used to set whether the column value contained in the control can be updated or inserted at the database or not. The default value is False.

TOEIntFloat.Reset

procedure Reset;

void Reset(**void**);

Description

This method is used to reset the control to the original value it contained. It is automatically called when the escape key is pressed when the control has the focus.

TOEDateTime

Hierarchy   Properties   Methods

Unit OVCL;

Description

This data-aware Visual Control is used to maintain date and time fields from a DataSource. It can handle one column of one row from the result set of a TDataSet at a time, hence it is called a single-row control.

TOEDateTime.DataField

property DataField: **String**;

property AnsiString DataField;

Description

This is used to set the column in the result set that the control must be populated with.

TOEDateTime.DataSource

property DataSource: TDataSource;

property TDataSource* DataSource;

Description

This is used to set the TDataSource component that links the control with a result set.

TOEDateTime.Field

property Field: TField;

property TField* Field;

Description

This returns the column TField object that the control is associated with.

TOEDateTime.ReadOnly

property ReadOnly: **Boolean**;

property bool ReadOnly;

Description

This is used to set whether the column value contained in the control can be updated or inserted at the database or not. The default value is False.

TOEDateTime.Reset

procedure Reset;

void Reset(**void**);

Description

This method is used to reset the control to the original value it contained. It is automatically called when the escape key is pressed when the control has the focus.

TDSComboBox

Hierarchy Properties Methods

Unit OVCL;

Description

This control lists all the available ODBC DataSources on the local machine when the Populate method is called.

TDSComboBox.DataSource

property DataSource: **String**;

property AnsiString DataSource;

Description

Used to retrieve the current DataSource selected in the component, and to set the current DataSource displayed in the component.

TDSComboBox.DataSourceType

TDataSourceType = (dsDefault, dsUser, dsSystem);

property DataSourceType: TDataSourceType;

TDataSourceType {dsDefault, dsUser, dsSystem};

property TDataSourceType DataSourceType;

Description

Specifies the type of DataSources to list in the component, according to the following table:

dsDefault	User and System DataSources.
dsUser	User DataSources.
dsSystem	System DataSources.

TDSComboBox.Populate

procedure Populate;

void Populate(**void**);

Description

Used to populate the component with the currently available DataSources or refresh the list of DataSources in the component.

TDSListBox

Hierarchy Properties Methods

Unit OVCL;

Description

This control lists all the available ODBC DataSources on the local machine when the Populate method is called.

TDSListBox.DataSource

property DataSource: **String**;

property AnsiString DataSource;

Description

Used to retrieve the current DataSource selected in the component, and to set the current DataSource displayed in the component.

TDSListBox.DataSourceType

TDataSourceType = (dsDefault, dsUser, dsSystem);

property DataSourceType: TDataSourceType;

TDataSourceType {dsDefault, dsUser, dsSystem};

property TDataSourceType DataSourceType;

Description

Specifies the type of DataSources to list in the component, according to the following table:

dsDefault	User and System DataSources.
dsUser	User DataSources.
dsSystem	System DataSources.

TDSListBox.Populate

procedure Populate;

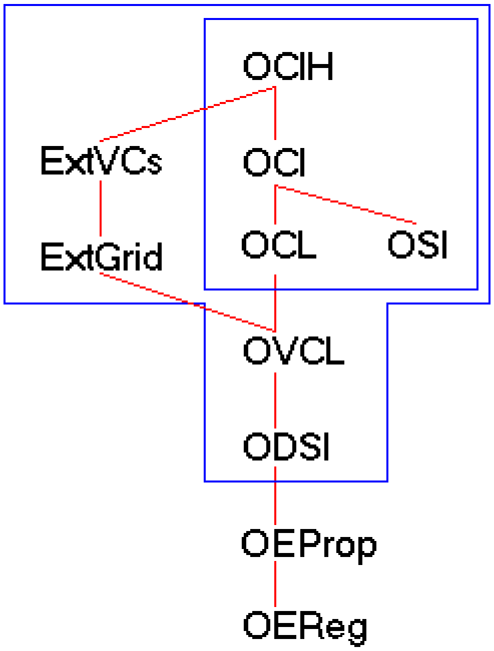
void Populate(**void**);

Description

Used to populate the component with the currently available DataSources or refresh the list of DataSources in the component.

Unit Structure

The ODBCExpress units are structured to optimize the application executable size, depending on the units used in the application. In the diagram below, the blue blocks indicate the two basic unit groups: the non-data-aware visual control group on the left and the ODBC constants and functions group on the right. When using non-visual ODBCExpress functionality, such as found in the OCL and OSI units, the inner blue block indicates the units that will be compiled into the application. When using visual ODBCExpress functionality, such as found in the OVCL and ODSI units, the outer blue block indicates the units that will be compiled into the application. The OEReg and OEProp units, which contains the Register procedure and all of the ODBCExpress propery editors, is only used at design time and won't be compiled into the application:



Arial Regular 14

TExtendedComboBox

Hierarchy Properties

Unit ExtVCs;

Description

The TExtendedComboBox allows you to display images for each item in the ComboBox. An ImageList is provided for the ComboBox and an ItemIndex is set for each item in the ComboBox using the Images property.

TExtendedComboBox.ImageList

property ImageList: TImageList;

property TImageList* ImageList;

Description

Specifies the list of images to display in the ComboBox.

TExtendedComboBox.Images

property Images: TComboImages;

property TComboImages* Images;

Description

Used to specify which image in the ImageList must be displayed for each item in the ComboBox.

TComboImages

Hierarchy Properties Methods

Unit ExtVCs;

Description

Used to specify which image in the ImageList must be displayed for each item in the ComboBox.

TComboImages.Add

```
procedure Add(S: String;
              ImageIndex: Integer);
```

```
void Add(AnsiString S,
      int ImageIndex);
```

Description

Used to add an item S to the ComboBox with an image of index ImageIndex in the ImageList.

TComboImages.AddObject

```
procedure AddObject(S: String;
                    ImageIndex: Integer;
                    AObject: TObject);
```

```
void AddObject(AnsiString S,
               int ImageIndex,
               TObject* AObject);
```

Description

Used to add an item S and object AObject to the ComboBox with an image of index ImageIndex in the ImageList.

TComboImages.Clear

procedure Clear;

void Clear(**void**);

Descripton

Used to clear the items in the ComboBox.

TComboImages.Delete

procedure Delete(Index: **Integer**);

void Delete(**int** Index);

Description

Used to delete the item at position Index in the ComboBox.

TComboImages.ImageIndex

property ImageIndex[Index: **Integer**]: **Integer**;

property int ImageIndex[**int** Index];

Description

Used to set or retrieve the index of the image in the ImageList displayed for the given item number Index in the ComboBox.

TComboImages.Insert

```
procedure Insert(Index: Integer;
                 S: String;
                 ImageIndex: Integer);
```

```
void Insert(int Index,
            AnsiString S,
            int ImageIndex);
```

Description

Used to insert an item S at position Index in the ComboBox with an image of index ImageIndex in the ImageList.

TComboImages.InsertObject

```
procedure InsertObject(Index: Integer;
                       S: String;
                       ImageIndex: Integer;
                       AObject: TObject);
```

void InsertObject(**int** Index,
AnsiString S,
int ImageIndex,
TObject* AObject);

Description

Used to insert an item S and object AObject at position Index in the ComboBox with an image of index ImageIndex in the ImageList.

TIntFloatPicker

Hierarchy Properties Methods Events

Unit ExtVCs;

Description

Used to edit integer and float numbers. The control will only allow valid characters to be entered. It also contains optional up-down spin buttons to change the value in the control.

TIntFloatPicker.Change

procedure Change; **virtual**;

virtual void Change(**void**);

Description

This method, which calls the OnChange event, is called each time the value in the control changes.

TIntFloatPicker.Decimals

property Decimals: **Integer**;

property int Decimals;

Description

This property is used to set the number of decimal places allowed when the control accepts float values. The default value is 2.

TIntFloatPicker.DropDown

procedure DropDown; **virtual**;

virtual void DropDown(**void**);

Description

This method, which calls the OnDropDown event, is called each time the up-down spin button is clicked.

TIntFloatPicker.Enabled

property Enabled: **Boolean**;

property bool Enabled;

Description

Used to enable or disable the control for user input. Set to True to enable the control, otherwise False. The default value is True.

TIntFloatPicker.Float

property Float: **Double**;

property double Float;

Description

Used to read and set the current float value contained in the control.

TIntFloatPicker.Int

property Int: **Integer**;

property int Int;

Description

Used to read and set the current integer value contained in the control.

TIntFloatPicker.KeyDown

procedure KeyDown(**var** Key: **Word**;
Shift: TShiftState); **virtual**;

virtual void KeyDown(**Word** &Key,
TShiftState Shift);

Description

This method, which calls the OnKeyDown event, is called each time a key is depressed when the control has focus.

TIntFloatPicker.KeyPress

procedure KeyPress(**var** Key: **Char**); **virtual**;

virtual void KeyPress(**char** &Key);

Description

This method, which calls the OnKeyPress event, is called each time a key is pressed when the control has focus.

TIntFloatPicker.KeyUp

procedure KeyUp(**var** Key: **Word**;
                  Shift: TShiftState); **virtual**;

virtual void KeyUp(**Word** &Key,
                  TShiftState Shift);

Description

This method, which calls the OnKeyUp event, is called each time a key is released when the control has focus.

TIntFloatPicker.Kind

TIntFloatKind = (ifkInt, ifkFloat);

property Kind: TIntFloatKind;

TIntFloatKind {ifkInt, ifkFloat};

property TIntFloatKind Kind;

Description

This property is used to set the control to accept either integer or float values. It can be one of the following two values:

ifkInt	Accept integer values.
ifkFloat	Accept float values.

The default value is ifkInt.

TIntFloatPicker.Max

property Max: **Integer**;

property int Max;

Description

Specifies the maximum value that may be entered in the control. If Max <= Min then any value can be entered in the control. The default value is 0.

TIntFloatPicker.Min

property Min: **Integer**;

property int Min;

Description

Specifies the minimum value that may be entered in the control. If Min >= Max then any value can be entered in the control. The default value is 0.

TIntFloatPicker.OnChange

property OnChange: TNotifyEvent;

property TNotifyEvent OnChange;

Description

This event is called by the Change method.

TIntFloatPicker.OnDropDown

property OnDropDown: TNotifyEvent;

property TNotifyEvent OnDropDown;

Description

This event is called by the DropDown method.

TIntFloatPicker.OnKeyDown

property OnKeyDown: TKeyEvent;

property TKeyEvent OnKeyDown;

Description

This event is called by the KeyDown method.

TIntFloatPicker.OnKeyPress

property OnKeyPress: TKeyPressEvent;

property TKeyPressEvent OnKeyPress;

Description

This event is called by the KeyPress method.

TIntFloatPicker.OnKeyUp

property OnKeyUp: TKeyEvent;

property TKeyEvent OnKeyUp;

Description

This event is called by the KeyUp method.

TIntFloatPicker.UpDown

property UpDown: **Boolean**;

property bool UpDown;

Description

This property is used to set whether the up-down spin buttons must be visible or not. The default value is True.

TExtendedListBox

Hierarchy Properties

Unit ExtVCs;

Description

The TExtendedListBox allows you to display images for each item in the ListBox. An ImageList is provided for the ListBox and an ItemIndex is set for each item in the ListBox using the Images property.

TExtendedListBox.ImageList

property ImageList: TImageList;

property TImageList* ImageList;

Description

Specifies the list of images to display in the ListBox.

TExtendedListBox.Images

property Images: TListImages;

property TListImages* Images;

Description

Used to specify which image in the ImageList must be displayed for each item in the ListBox.

TListImages

Hierarchy Properties Methods

Unit ExtVCs;

Description

Used to specify which image in the ImageList must be displayed for each item in the ListBox.

TListImages.Add

```
procedure Add(S: String;
              ImageIndex: Integer);
```

```
void Add(AnsiString S,
     int ImageIndex);
```

Description

Used to add an item S to the ListBox with an image of index ImageIndex in the ImageList.

TListImages.AddObject

```
procedure AddObject(S: String;
                    ImageIndex: Integer;
                    AObject: TObject);
```

```
void AddObject(AnsiString S,
               int ImageIndex,
               TObject* AObject);
```

Description

Used to add an item S and object AObject to the ListBox with an image of index ImageIndex in the ImageList.

TListImages.Clear

```
procedure Clear;
```

```
void Clear(void);
```

Description

Used to clear the items in the ListBox.

TListImages.Delete

```
procedure Delete(Index: Integer);
```

```
void Delete(int Index);
```

Description

Used to delete the item at position Index in the ListBox.

TListImages.ImageIndex

```
property ImageIndex[Index: Integer]: Integer;
```

```
property int ImageIndex[int Index];
```

Description

Used to set or retrieve the index of the image in the ImageList displayed for the given item number Index in the ListBox.

TListImages.Insert

```
procedure Insert(Index: Integer;
                 S: String;
                 ImageIndex: Integer);
```

```
void Insert(int Index,
            AnsiString S,
            int ImageIndex);
```

Description

Used to insert an item S at position Index in the ListBox with an image of index ImageIndex in the ImageList.

TListImages.InsertObject

```
procedure InsertObject(Index: Integer;
                       S: String;
                       ImageIndex: Integer;
                       AObject: TObject);
```

```
void InsertObject(int Index,
```

AnsiString S,
int ImageIndex,
TObject* AObject);

Description

Used to insert an item S and object AObject at position Index in the ListBox with an image of index ImageIndex in the ImageList.

TExtendedGrid

Hierarchy Properties Methods Events

Unit ExtGrid;

Description

The ExtendedGrid is the most feature rich if the Visual Components. It allows you to set different properties for each column of the Grid. For example, you can format the text of a column heading and the column contents, you can specify the editability of each column, and you can fill a column with controls such as CheckBoxes, ComboBoxes and even pictures.

TExtendedGrid.AddRow

procedure AddRow;

void AddRow(**void**);

Description

This method will add a new row to the end of the grid.

TExtendedGrid.CanControlShow

function CanControlShow(ACol, ARow: **Longint**): **Boolean**; **virtual**;

virtual bool CanControlShow(**long** ACol,
long ARow);

Description

The result of this method indicates if the control in column ACol and row ARow is editable or not. If the control in column ACol is an Edit control, it will return the result of the CanEditShow method.

TExtendedGrid.CanEditShow

function CanEditShow: **Boolean**; **override**;

virtual bool CanEditShow(**void**);

Description

The result of this method indicates if the Edit control in column Col and row Row is editable or not.

TExtendedGrid.CellString

property CellString[ACol, ARow: **Longint**]: **String**;

property AnsiString CellString[**long** ACol][**long** ARow];

Description

Returns the value in cell ACol, ARow of the Grid as a string. It will be one of the following values, depending on the Columns[ACol].ContentsControl property:

cImage	PictureValue[ACol,ARow]
cCheckBox	CheckValue[ACol,ARow]
cComboBox	ComboValue[ACol,ARow]
otherwise	Cells[ACol,ARow]

TExtendedGrid.CheckValue

property CheckValue[ACol, ARow: **Longint**]: **String**;

property AnsiString CheckValue[**long** ACol][**long** ARow];

Description

Used to read or write the value of the CheckBox in cell ARow, ACol of the Grid. The value which put the CheckBox in its current state (checked, unchecked or grayed) will be returned. If a new value is assigned to this property, the value of the CheckBox is set as follows (in this order):

Checked	if (NewValue = OnValue) or (OnValue = '' and NewValue <> OffValue)
Unchecked	if (NewValue = OffValue) or (OffValue = '' and NewValue <> OnValue)
Grayed	if (NewValue <> OnValue) and (NewValue <> OffValue)

TExtendedGrid.Clear

procedure Clear;

void Clear(**void**);

Description

This method will clear all the rows from the Grid, after which it will clear the cells of the single row left in the Grid.

TExtendedGrid.ClearRow

procedure ClearRow(Index: **Longint**);

void ClearRow(**long** Index);

Description

This method will clear the cells of row number Index in the Grid.

TExtendedGrid.Columns

property Columns: TGridColumns;

property TGridColumns* Columns;

Description

Manages all the properties of the columns in the ExtendedGrid.

A design time property editor exists for this property, and can be accessed via the Object Inspector. A defined column can be moved by dragging the corresponding row header of the defined column in the property editor.

TExtendedGrid.ComboValue

property ComboValue[ACol, ARow: **Longint**]: **String**;

property AnsiString ComboValue[**long** ACol][**long** ARow];

Description

Used to read or write the value of the ComboBox in cell ARow, ACol of the Grid. The value which put the ComboBox in its current state will be returned. If a new value is assigned to this property, the value of the ComboBox is set as follows (in this order):

To item number n	if NewValue = corresponding value of item number n, n >= -1.
To item number n	if corresponding value of item number n = '', n >= -1
To item number -1 with value NewValue	if Columns[ACol].ComboEditable = True
To item number -1 with value ''	if Columns[ACol].ComboEditable = False

TExtendedGrid.CopyRow

procedure CopyRow(FromIndex, ToIndex: **Longint**);

void CopyRow(**long** FromIndex,
long ToIndex);

Description

This method will copy the cells of row number FromIndex to row number ToIndex.

TExtendedGrid.DeleteRow

procedure DeleteRow(Index: **Longint**);

void DeleteRow(**long** Index);

Description

This method will delete row number Index from the Grid.

TExtendedGrid.DoControlChanged

procedure DoControlChanged(ACol, ARow: **Longint**); **virtual**;

virtual void DoControlChanged(**long** ACol,
                                 long ARow);

Description

This method is called when the value in the control of cell ACol, ARow changed. It will call the OnControlChanged event.

TExtendedGrid.DoEditChanged

procedure DoEditChanged(ACol, ARow: **Longint**); **virtual**;

virtual void DoEditChanged(**long** ACol,
                              long ARow);

Description

This method is called when the value in the Edit control of cell ACol, ARow changed. It will call the DoControlChanged method.

TExtendedGrid.DoResize

procedure DoResize(Sender: TObject); **virtual**;

virtual void DoResize(TObject* Sender);

Description

This method is called when the Grid is resized.

TExtendedGrid.ExchangeRows

procedure ExchangeRows(Index1, Index2: **Longint**);

void ExchangeRow(**long** Index1,
                    long Index2);

Description

This method will exchange the positions of two rows in the Grid.

TExtendedGrid.InsertRow

procedure InsertRow(Index: **Longint**);

void InsertRow(**long** Index);

Description

This method will insert a new row at position Index in the Grid.

TExtendedGrid.MoveRow

procedure MoveRow(Index, NewIndex: **Longint**);

```
void MoveRow(long Index,
             long NewIndex);
```

Description

This method will move a row from one position in the Grid to another.

TExtendedGrid.OnControlChanged

```
TControlEvent = procedure (Sender: TObject;
                           ACol, ARow: Longint) of object;
```

property OnControlChanged: TControlEvent;

```
void (closure* TControlEvent)(TObject* Sender,
                              long ACol,
                              long ARow);
```

property TControlEvent OnControlChanged;

Description

This event is called when the value in the control of cell ACol, ARow changed.

TExtendedGrid.PictureValue

property PictureValue[ACol, ARow: **Longint**]: **String**;

property AnsiString PictureValue[**long** ACol][**long** ARow];

Description

Used to read or write the value of the picture resource in cell ARow, ACol of the Grid. The value which put the cell in its current state will be returned. If a new value is assigned to this property, the value of the picture resource is set as follows (in this order):

To item number n	if NewValue = corresponding value of item number n, n >= -1.
To item number n	if corresponding value of item number n = '', n >= -1.
To the value ''	

TGridColumns

Hierarchy Properties Methods Events

Unit ExtGrid;

Description

Manages all the properties of the columns in the ExtendedGrid.

TGridColumns.AddItem

procedure AddItem;

void AddItem(**void**);

Description

Used to add a column of type TGridColumn to the Grid.

TGridColumns.DeleteItem

procedure DeleteItem(Index: **Integer**);

void DeleteItem(**int** Index);

Description

Used to delete a column of type TGridColumn from position Index in the Grid.

TGridColumns.DoChange

procedure DoChange(Sender: TObject); **virtual**;

virtual void DoChange(TObject* Sender);

Description

This method is called when a property value in one of the columns of the Grid changes. It calls the OnChange event.

TGridColumns.ExchangeItems

procedure ExchangeItems(Index1, Index2: **Integer**);

void ExchangeItems(**int** Index1,
                                int Index2);

Description

Used to exchange the columns of type TGridColumn at positions Index1 and Index2 in the Grid.

TGridColumns.InsertItem

procedure InsertItem(Index: **Integer**);

void InsertItem(**int** Index);

Description

Used to insert a column of type TGridColumn at position Index in the Grid.

TGridColumns.ItemCount

property ItemCount: **Integer**;

property int ItemCount;

Description

Specifies the number of columns in the Grid. It contains the same value as the ColCount property of the Grid.

TGridColumns.Items

property Items[Index: **Integer**]: TGridColumn;

property TGridColumn* Items[**int** Index];

Description

The Items property contains the list of columns of type TGridColumn in the Grid. Since this is the default property, you can reference a column in the Grid as Columns[i] instead of Columns.Items[i].

TGridColumns.MoveItem

procedure MoveItem(Index, NewIndex: **Integer**);

void MoveItem(**int** Index,
                        int NewIndex);

Description

Used to move a column of type TGridColumn from position Index to position NewIndex in the Grid.

TGridColumns.OnChange

property OnChange: TNotifyEvent;

property TNotifyEvent OnChange;

Description

This event is called when a property value in one of the columns of the Grid changes.

TGridColumn

Hierarchy Properties Methods Events

Unit ExtGrid;

Description

Manages the properties for each column in the ExtendedGrid.

TGridColumn.CheckOffValue

property CheckOffValue: **String**;

property AnsiString CheckOffValue;

Description

Used to specify which value will set the state of the CheckBox to Unchecked. If CheckOffValue is an empty string, the state of the CheckBox will be set to Unchecked for all values not equal to CheckOnValue. The default value is the string '0'.

TGridColumn.CheckOnValue

property CheckOnValue: **String**;

property AnsiString CheckOnValue;

Desctiption

Used to specify which value will set the state of the CheckBox to Checked. If CheckOnValue is an empty string, the state of the CheckBox will be set to Checked for all values not equal to CheckOffValue. The default value is an empty string.

TGridColumn.ComboEditable

property ComboEditable: **Boolean**;

property bool ComboEditable;

Description

Used to specify whether you are allowed to enter in a new value into the ComboBox, or if you are just allowed to select a new value from the dropdown list. If you want to allow a new value to be entered into the ComboBox, set this property to True, otherwise to False. The default value is False.

TGridColumn.ComboFilled

property ComboFilled: **Boolean**;

property bool ComboFilled;

Description

Used to determine if the ComboBox is pre-filled or not. If True, then the ComboItems and ComboValues properties must also be set. If False, then the ComboStatement property must also be set. The default value is True.

TGridColumn.ComboItems

property ComboItems: TStringList;

property TStringList* ComboItems;

Description

Contains the items that the ComboBox is filled with. For each item, there must be a corresponding value in the ComboValues property. When a value is passed to the ComboBox, this value is looked up in the ComboValues property, and then the ComboBox is set to the corresponding item in the ComboItems property. If this value is not found in the ComboValues property, the ItemIndex of the ComboBox is set to -1. By default it contains no values.

TGridColumn.ComboStatement

property ComboStatement: **String**;

property AnsiString ComboStatement;

Description

Used to set a SQL statement for the ComboBox. The first column in the statement is used to fill the ComboItems with, while the second column in the statement is used to fill the ComboValues with. The default value is an empty string.

TGridColumn.ComboValues

property ComboValues: TStringList;

property TStringList* ComboValues;

Description

Contains the values used to set the ComboBox. For each value, there must be a corresponding item in the ComboItems property. When a value is passed to the ComboBox, this value is looked up in the ComboValues property, and then the ComboBox is set to the corresponding item in the ComboItems property. If this value is not found in the ComboValues property, the ItemIndex of the ComboBox is set to -1. By default it contains no values.

TGridColumn.ContentsClipStyle

TClipStyle = (csClip, csEllips, csWrap);

property ContentsClipStyle: TClipStyle;

TClipStyle {csClip, csEllips, csWrap};

property TClipStyle ContentsClipStyle;

Description

Used to set the clipping style of the column's contents text. The default value is csClip. It can be set to one of the following values:

csClip	Cuts off text which does not fit into a cell.
csEllipse	Puts an ellipses '...' next to the text which does not fit into a cell.
csWrap	Word wraps the text that does not fit into a cell.

TGridColumn.ContentsControl

TControlType = (cEdit, cImage, cCheckBox, cComboBox, cEditFloat, cEditNum, cSpinNum, cSpinDate, cSpinTime);

property ContentsControl: TControlType;

TControlType {cEdit, cImage, cCheckBox, cComboBox, cEditFloat, cEditNum, cSpinNum, cSpinDate, cSpinTime};

property TControlType ContentsControl;

Description

Used to set the control displayed in the column. The default value is cEdit. It can be set to one of the following values:

cEdit	Display an Edit control in the column.
cImage	Display an Image control in the column.

cCheckBox	Display a CheckBox control in the column.
cComboBox	Display a ComboBox control in the column.
cEditFloat	Display an EditFloat control in the column.
cEditNum	Display an EditNum control in the column.
cSpinNum	Display a SpinNum control in the column.
cSpinDate	Display a SpinDate control in the column.
cSpinTime	Display a SpinTime control in the column.

TGridColumn.ContentsFont

property ContentsFont: TFont;

property TFont* ContentsFont;

Description

Used to set the font of the column's contents text.

TGridColumn.ContentsHAlign

THAlign = (haLeft, haCenter, haRight);

property ContentsHAlign: THAlign;

THAlign {haLeft, haCenter, haRight};

property THAlign ContentsHAlign;

Description

Used to set the horizontal alignment of the column's contents text. The default value is haLeft. It can be set to one of the following values:

haLeft	Align the text to the left of a cell.
haCenter	Position the text in the center of a cell.
haRight	Align the text to the right of a cell.

TGridColumn.ContentsVAlign

TVAlign = (vaTop, vaMiddle, vaBottom);

property ContentsVAlign: TVAlign;

TVAlign {vaTop, vaMiddle, vaBottom};

property TVAlign ContentsVAlign;

Description

Used to set the vertical alignment of the column's contents text. The default value is vaMiddle. It can be set to one of the following values:

vaTop	Align the text to the top of a cell.
vaMiddle	Position the text in the center of a cell.
vaBottom	Align the text to the bottom of a cell.

TGridColumn.DoChange

procedure DoChange(Sender: TObject); **virtual**;

virtual void DoChange(TObject* Sender);

Description

This method is called when a property in the column changes. It calls the OnChange event.

TGridColumn.Editable

property Editable: **Boolean**;

property bool Editable;

Description

Used to specify whether the column is editable or not. However, if the goEditing Option of the Grid is set to False, then this property is ignored. The default value is False.

TGridColumn.HeaderBevelWidth

property HeaderBevelWidth: **Integer**;

property int HeaderBevelWidth;

Description

Used to set the bevel width of the column's header. The default value is 1.

TGridColumn.HeaderBorderWidth

property HeaderBorderWidth: **Integer**;

property int HeaderBorderWidth;

Description

Used to set the bevel border width of the column's header. The default value is 0.

TGridColumn.HeaderClipStyle

TClipStyle = (csClip, csEllips, csWrap);

property HeaderClipStyle: TClipStyle;

TClipStyle {csClip, csEllips, csWrap};

property TClipStyle HeaderClipStyle;

Description

Used to set the clipping style of the column's header text. The default value is csClip. It can be set to one of the following values:

csClip	Cuts off text which does not fit into a cell.
csEllipse	Puts an ellipses '...' next to the text which does not fit into a cell.
csWrap	Word wraps the text that does not fit into a cell.

TGridColumn.HeaderFont

property HeaderFont: TFont;

property TFont* HeaderFont;

Description

Used to set the font of the column's header text.

TGridColumn.HeaderHAlign

THAlign = (haLeft, haCenter, haRight);

property HeaderHAlign: THAlign;

THAlign {haLeft, haCenter, haRight};

property THAlign HeaderHAlign;

Description

Used to set the horizontal alignment of the column's header text. The default value is haLeft. It can be set to one of the following values:

haLeft	Align the text to the left of a cell.
haCenter	Position the text in the center of a cell.
haRight	Align the text to the right of a cell.

TGridColumn.HeaderInnerBevel

property HeaderInnerBevel: TPanelBevel;

property TPanelBevel HeaderInnerBevel;

Description

Used to set the inner bevel of the column's header. The default value is bvNone. It can be set to one of the following values:

bvNone	No inner bevelling of a cell.
bvLowered	Lower the inner bevel of a cell.
bvRaised	Raise the inner bevel of a cell.

TGridColumn.HeaderOuterBevel

property HeaderOuterBevel: TPanelBevel;

property TPanelBevel HeaderOuterBevel;

Description

Used to set the outer bevel of the column's header. The default value is bvNone. It can be set to one of the following values:

bvNone	No outer bevelling of a cell.
bvLowered	Lower the outer bevel of a cell.
bvRaised	Raise the outer bevel of a cell.

TGridColumn.HeaderVAlign

TVAlign = (vaTop, vaMiddle, vaBottom);

property HeaderVAlign: TVAlign;

TVAlign {vaTop, vaMiddle, vaBottom};

property TVAlign HeaderVAlign;

Description

Used to set the vertical alignment of the column's header text. The default value is vaMiddle. It can be set to one of the following values:

vaTop	Align the text to the top of a cell.
vaMiddle	Position the text in the center of a cell.
vaBottom	Align the text to the bottom of a cell.

TGridColumn.OnChange

property OnChange: TNotifyEvent;

property TNotifyEvent OnChange;

Description

This event is called when a property in the column changes.

TGridColumn.PictureItems

property PictureItems: TStringList;

property TStringList* PictureItems;

Description

Contains the resource names of the images to display. For each item, there must be a corresponding value in the PictureValues property. When a value is passed to the PictureValue property of the Grid, this value is looked up in the PictureValues property, and then the corresponding resource in the PictureItems property is displayed. If this value is not found in the PictureValues property, nothing is displayed. By default it contains no values.

TGridColumn.PictureValues

property PictureValues: TStringList;

property TStringList* PictureValues;

Description

Contains the values used to display an image. For each value, there must be a corresponding item in the PictureItems property. When a value is passed to the PictureValue property of the Grid, this value is looked up in the PictureValues property, and then the corresponding resource in the PictureItems property is displayed. If this value is not found in the PictureValues property, nothing is displayed. By default it contains no values.

TGridColumn.Title

property Title: **String**;

property AnsiString Title;

Description

Used to set the title text of the column. The default value is an empty string.

TGridColumn.Width

property Width: **Integer**;

property int Width;

Description

Used to set the width of the column. The column widths can also be set using the ColWidths property of the Grid. The default value is 64.

Conversion Functions

The following utility functions can be used to do data type manipulation:

PhysSize
SqlTypeToColType
ColTypeToSqlType
ToValue
ToString
ToDouble
ToInteger
ToTimeStamp

PhysSize

function PhysSize(CType: SQLSMALLINT): **Word**;

Word PhysSize(SQLSMALLINT CType);

Description

Returns the physical size (in bytes) of the given Storage Type.

SqlTypeToColType

function SqlTypeToColType(SqlType: SQLSMALLINT): SQLSMALLINT;

SQLSMALLINT SqlTypeToColType(SQLSMALLINT SqlType);

Description

Returns the Storage Type for the given Sql Type, as specified by the ODBC Constants Map. In the case where there are two Storage Types for a Sql Type, the signed Storage Type is returned.

ColTypeToSqlType

function ColTypeToSqlType(ColType: SQLSMALLINT): SQLSMALLINT;

SQLSMALLINT ColTypeToSqlType(SQLSMALLINT ColType);

Description

Returns the Sql Type for the given Storage Type, as specified by the ODBC Constants Map. In the case where there are more than one Sql Type for a Storage Type, only one specific Sql Type is returned.

This function is used to provide default mappings for the binding of parameters in a SQL statement.

ToValue

```
function ToValue(SValue: String;
                 CValue: SQLPOINTER;
                 CType: SQLSMALLINT
                 StringTrimming: TStringTrimming): Boolean;
```

```
bool ToValue(AnsiString SValue,
             SQLPOINTER CValue,
             SQLSMALLINT CType,
             TStringTrimming StringTrimming);
```

Description

Takes a string SValue, converts it and places it in the existing storage pointed to by CValue, of type CType. It applies the StringTrimming option to the string before converting it.

ToString

```
function ToString(CValue: SQLPOINTER;
                  CType: SQLSMALLINT;
                  StringTrimming: TStringTrimming): String;
```

```
AnsiString ToString(SQLPOINTER CValue,
                    SQLSMALLINT CType,
                    TStringTrimming StringTrimming);
```

Description

Returns a String value corresponding to the existing storage pointed to by CValue, of type CType. It applies the StringTrimming option to the string before returning it.

ToDouble

```
function ToDouble(CValue: SQLPOINTER;
                  CType: SQLSMALLINT): Double;
```

```
double ToDouble(SQLPOINTER CValue,
                SQLSMALLINT CType);
```

Description

Returns a Double value corresponding to the existing storage pointed to by CValue, of type CType.

ToInteger

```
function ToInteger(CValue: SQLPOINTER;
                   CType: SQLSMALLINT): Integer;
```

```
int ToInteger(SQLPOINTER CValue,
              SQLSMALLINT CType);
```

Description

Returns an Integer value corresponding to the existing storage pointed to by CValue, of type CType.

ToTimeStamp

```
function ToTimeStamp(CValue: SQLPOINTER;
                     CType: SQLSMALLINT): TTimeStamp;

TTimeStamp ToTimeStamp(SQLPOINTER CValue,
                       SQLSMALLINT CType);
```

Description

Returns a TTimeStamp value corresponding to the existing storage pointed to by CValue, of type CType.

TimeStamp Functions

The following utility functions can be used to manipulate the ODBC date and time data types:

TimeStamp
TSEqual
TSCompare
TSToDateTime
DateTimeToTS
NullTS
TSNull
TSAsDate
DateAsTS
TSAsTime
TimeAsTS

TimeStamp

```
function TimeStamp(Year: SQLSMALLINT;
                   Month: SQLUSMALLINT;
                   Day: SQLUSMALLINT;
                   Hour: SQLUSMALLINT;
                   Minute: SQLUSMALLINT;
                   Second: SQLUSMALLINT;
                   Fraction: SQLUINTEGER): TTimeStamp;

TTimeStamp TimeStamp(SQLSMALLINT Year;
                     SQLUSMALLINT Month;
                     SQLUSMALLINT Day;
                     SQLUSMALLINT Hour;
                     SQLUSMALLINT Minute;
                     SQLUSMALLINT Second;
```

SQLUINTEGER Fraction);

Description

Takes the 7 fields of a TTimeStamp record as parameters and return a TTimeStamp record which contains these field values.

TSEqual

function TSEqual(TS1, TS2: TTimeStamp): **Boolean**;

bool TSEqual(TTimeStamp TS1,
TTimeStamp TS2);

Description

Returns True if the TTimeStamp records TS1 and TS2 are the same, otherwise returns False.

TSCompare

function TSCompare(TS1, TS2: TTimeStamp): **Integer**;

int TSCompare(TTimeStamp TS1,
TTimeStamp TS2);

Description

Returns one of the following values for the given TTimeStamp records TS1 and TS2:

-1	if TS1 is smaller than TS2
0	if TS1 equals TS2
1	if TS1 is larger than TS2

TSToDateTime

function TSToDateTime(TS: TTimeStamp): TDateTime;

TDateTime TSToDateTime(TTimeStamp TS);

Description

Returns a DateTime for a given TTimeStamp record.

DateTimeToTS

function DateTimeToTS(DateTime: TDateTime): TTimeStamp;

TTimeStamp DateTimeToTS(TDateTime DateTime);

Description

Returns a TTimeStamp record for a given DateTime.

NullTS

function NullTS: TTimeStamp;

TTimeStamp NullTS(**void**);

Description

Returns the null TTimeStamp value of DateTimeToTS(0).

TSNull

function TSNull(TS: TTimeStamp): **Boolean**;

bool TSNull(TTimeStamp TS);

Description

Returns True if the TTimeStamp record TS has the value NullTS, otherwise returns False.

TSAsDate

function TSAsDate(TS: TTimeStamp): TDate;

TDate TSAsDate(TTimeStamp TS);

Description

This method is used to convert an ODBC TTimeStamp to an ODBC TDate.

DateAsTS

function DateAsTS(D: TDate): TTimeStamp;

TTimeStamp DateAsTS(TDate D);

Description

This method is used to convert an ODBC TDate to an ODBC TTimeStamp.

TSAsTime

function TSAsTime(TS: TTimeStamp): TTime;

TTime TSAsTime(TTimeStamp TS);

Description

This method is used to convert an ODBC TTimeStamp to an ODBC TTime.

TimeAsTS

function TimeAsTS(T: TTime): TTimeStamp;

TTimeStamp TimeAsTS(TTime T);

Description

This method is used to convert an ODBC TTime to an ODBC TTimeStamp.

ODBC Constants Map

When data is retrieved from a DataSource, ODBCExpress creates storage for the data on the fly. For each Sql Type field in the database, the corresponding memory of type Storage Type is created, according to the following table. This table also lists all the possible values for Sql Type and Storage Type.

The following constants are declared in the unit OCIH:

Sql Type Constant	Storage Type Constant	ODBC Type
SQL_CHAR	SQL_C_CHAR	NullString
SQL_VARCHAR	SQL_C_CHAR	NullString
SQL_LONGVARCHAR	SQL_C_CHAR	SQLPOINTER
SQL_BINARY	SQL_C_BINARY	NullString
SQL_VARBINARY	SQL_C_BINARY	NullString
SQL_LONGVARBINARY	SQL_C_BINARY	SQLPOINTER
SQL_REAL	SQL_C_FLOAT	SQLREAL
SQL_DOUBLE	SQL_C_DOUBLE	SQLDOUBLE
SQL_FLOAT	SQL_C_DOUBLE	SQLDOUBLE
SQL_DECIMAL	SQL_C_DOUBLE	SQLDOUBLE
SQL_NUMERIC	SQL_C_DOUBLE	SQLDOUBLE
SQL_BIT	SQL_C_BIT	SQLCHAR
SQL_TINYINT	SQL_C_STINYINT	SQLSCHAR
	SQL_C_UTINYINT	SQLCHAR
SQL_SMALLINT	SQL_C_SSHORT	SQLSMALLINT
	SQL_C_USHORT	SQLUSMALLINT
SQL_INTEGER	SQL_C_SLONG	SQLINTEGER
	SQL_C_ULONG	SQLUINTEGER
SQL_BIGINT	SQL_C_SBIGINT	SQLBIGINT
	SQL_C_UBIGINT	SQLUBIGINT
SQL_TYPE_DATE	SQL_C_TYPE_DATE	TDate
SQL_TYPE_TIME	SQL_C_TYPE_TIME	TTime
SQL_TYPE_TIMESTAMP	SQL_C_TYPE_TIMESTAMP	TTimeStamp

Data Type Ranges

This table maps the data type ranges of ODBC Sql Type Constants.

The following constants are declared in the unit OCIH:

Sql Type Constant	**ODBC Range**
SQL_CHAR	1..255 bytes
SQL_VARCHAR	1..255 bytes
SQL_LONGVARCHAR	1..2^31 - 2 bytes
SQL_BINARY	1..256 bytes
SQL_VARBINARY	1..256 bytes
SQL_LONGVARBINARY	1..2^31 - 1 bytes
SQL_REAL	10^-38..10^38 (4 bytes)
SQL_DOUBLE	10^-308..10^308 (8 bytes)
SQL_FLOAT	10^-308..10^308 (8 bytes)
SQL_DECIMAL	10^-308..10^308 (8 bytes)
SQL_NUMERIC	10^-308..10^308 (8 bytes)
SQL_BIT	1 bit
SQL_TINYINT	Signed: -128..127 Unsigned: 0..255 (1 byte)
SQL_SMALLINT	Signed: -32768..32767 Unsigned: 0..65535 (2 bytes)
SQL_INTEGER	Signed: -2^31..2^31 - 1 Unsigned: 0..2^32 - 1 (4 bytes)
SQL_BIGINT	Signed: -2^63..2^63 - 1 Unsigned: 0..2^64 - 1 (8 bytes)