Script Executive

The Script Executive can be used to view, edit and execute SQL scripts. It can be started from the IBExpert Tools menu, using the respective icon in the Tools toolbar or [Ctrl + F12]. It is used for SQLs covering several rows. The Script Executive can both read and execute scripts.

Although Firebird/InterBase® can also process such procedure definitions in the SQL Editor, it is recommended using the Script Executive for more complex work, as it can do much more than the SQL Editor. There is a wealth of script language extensions including conditional directives, and it can also be used for executing multiple scripts from a single script. The main advantage of the Script Executive is that it displays all DDL and DML scripts of a connected database.

IBExpert version 2020.05.10 introduced support of the SET DBTRIGGERS ON/OFF directive. If DBTRIGGERS is OFF all consequent CONNECT/RECONNECT statements will be executed with the isc_dbp_no_db_triggers flag. Default value is ON. The “Add CONNECT statement” feature adds SET DBTRIGGERS OFF automatically if the “Supress database triggers” option is specified in a database registration record.

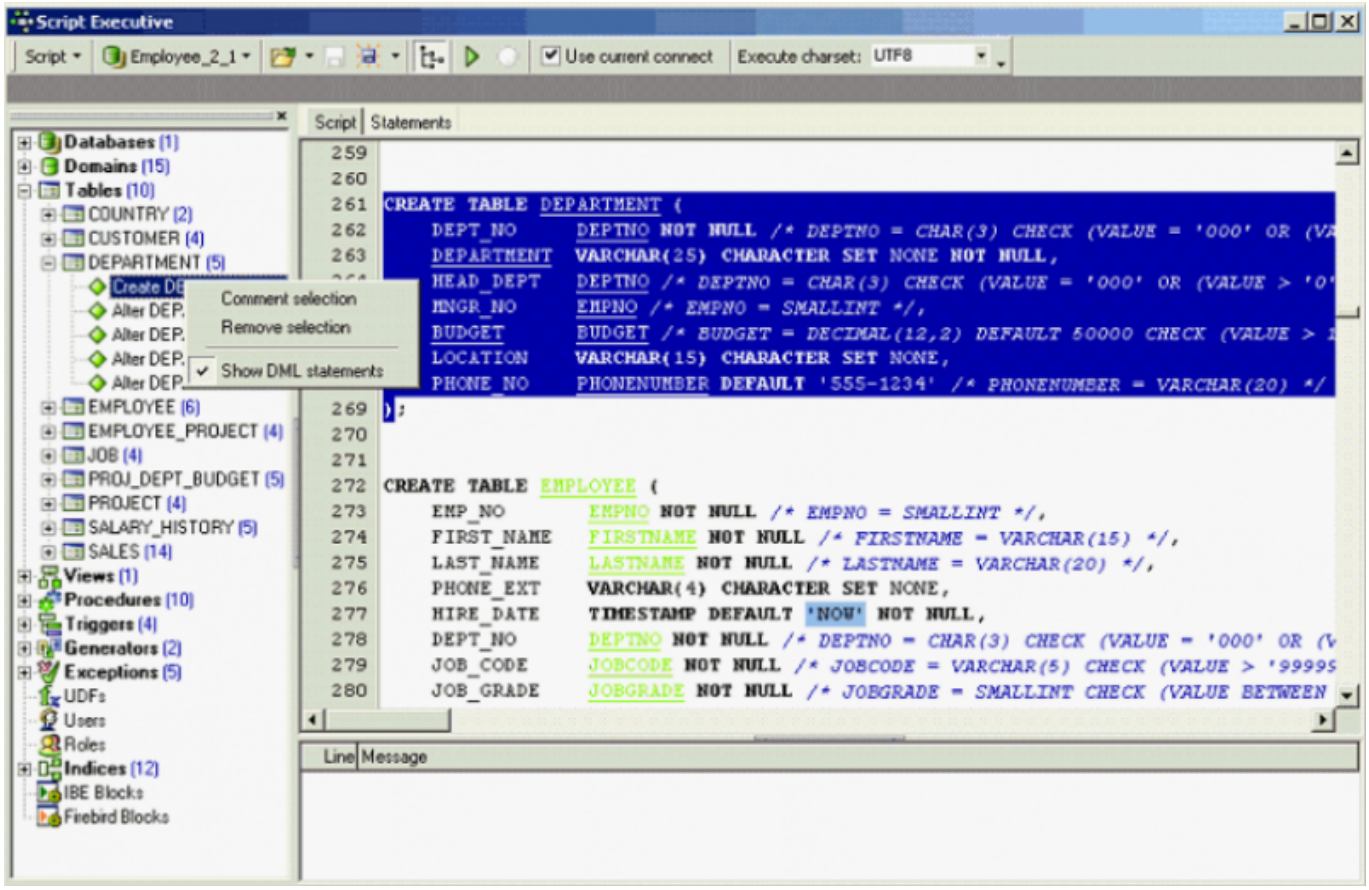
Script Explorer

The *Script Explorer* (the left-hand panel) displays all database objects, as well as IBEBlocks and Firebird blocks, used in the current script in a tree structure. It even allows you to find individual statements rapidly by clicking on the object in the tree. The *Script Explorer* can be blended in and out using the respective icon on the Script Executive toolbar. SQL scripts can be loaded from and saved to file if wished.

Objects may be dragged and dropped from the DB Explorer and SQL Assistant into the code editor window. When an object node(s) is dragged from the DB Explorer or SQL Assistant, IBExpert will offer various versions of text to be inserted into the code editor. It is also possible to customize the highlighting of variables. Use the IBExpert Options menu item, Editor Options / Colors to choose color and font style for variables.

Script page

Complete scripts can be transferred from the SQL Editor or extracted directly from the Extract Metadata Editor into the Script Executive using the relevant menu items (please refer directly to these subjects for further details).



Please note that the Script Executive always uses the default client library specified in the IBExpert Options menu item Environment Options / Preferences under *Default Client Library*, unless it is overridden using the SET CLIENTLIB command.

DML statements can be displayed in the Script Explorer tree. Simply right-click to open the context-sensitive menu and check/uncheck as wished.

The toolbar menu item, *Execute charset* offers the options *ANSI*, *UTF8* or Ask me. IBExpert offers full Unicode support. The internal representation of all texts in the code editors is Windows Unicode (UTF-16LE, two bytes per character). This allows you to use multilingual characters in your procedures, queries, database object descriptions etc., if you use the UTF8 character set when connecting to your database.

There is also support for *Before/After metadata change* events. Execution of DDL statements will fire before and after metadata event blocks if they are assigned and the *Use current connection* option is *ON*.

The following features are also available for the Script Executive and IBEScript:

- When no password and/or user name are specified in the CONNECT or CREATE DATABASE statements, a login dialog will appear.
- Case-sensitive user and role names (Firebird 3, Firebird 4) in CREATE DATABASE and CONNECT statements should be quoted with double quotes, e.g. CONNECT ... USER “SYSDBA” ROLE “Admin_Role” ... ;
- It is now also possible to change the connection character set (SET NAMES) and garbage collection option (SET GARBAGE_COLLECT) before the RECONNECT statement. Any SET commands mentioned which are followed by a RECONNECT statement will affect the new

connection.

The IBECurrentScriptPath environment variable is initialized internally before executing the script, and its value represents a path to the current script file (if the script is loaded/executed from a file). Typically you can use this variable in INPUT, SET BLOBFILE and SET PARAMFILE directives to specify where IBExpert/IBEScript should search for required files.

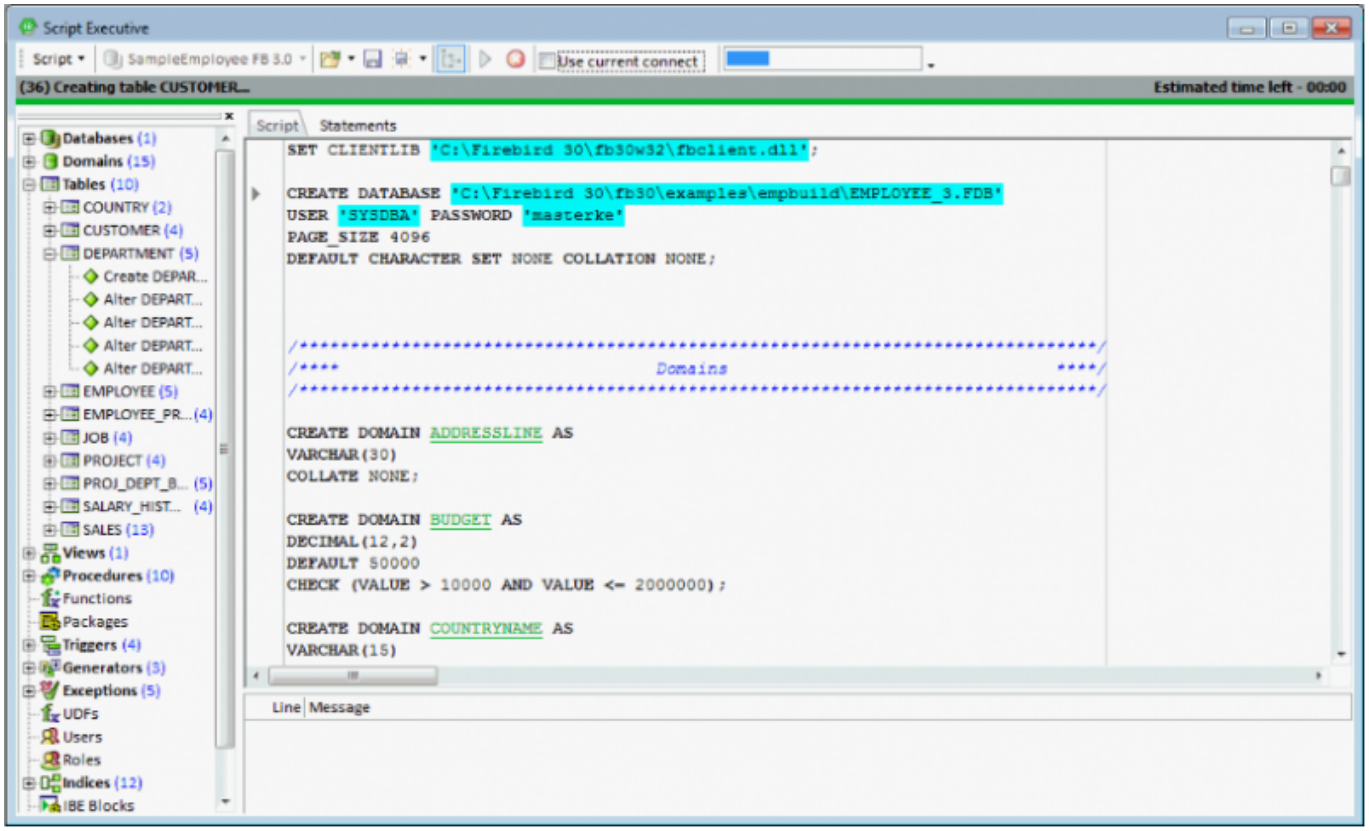
Example:

```
INPUT 'Inputs\data.sql';
...
SET BLOBFILE 'Data\blobs.lob';
```

When you're working with a database using the UTF8 character set IBExpert performs automatic conversion from UTF8 to Windows Unicode (when opening) and back (when you compile). This applies to Firebird 2.1 and 2.5 databases. For other databases you need to enable this behavior manually (if you really need this!) by flagging the *Do NOT perform conversion from/to UTF8* checkbox in the Database Registration Info. As a rule, IBExpert knows when it must convert strings from Windows Unicode to UTF8 but sometimes it is necessary to specify the conversion type manually. This allows you to specify the necessary charset manually.

The Script page includes other features, such as code completion (please refer to Code Insight for details) and [F1] context-sensitive keyword help - both familiar from the SQL Editor. The SQL Editor menu can be called by right-clicking in the script area.

The *Advanced progess* option: the progress bar is shown on the Script Executive toolbar, and if this option is enabled (default), IBExpert roughly calculates and displays the remaining time using known script size and time already spent, and scrolls the script text editor accordingly to the part of the script currently being executed.



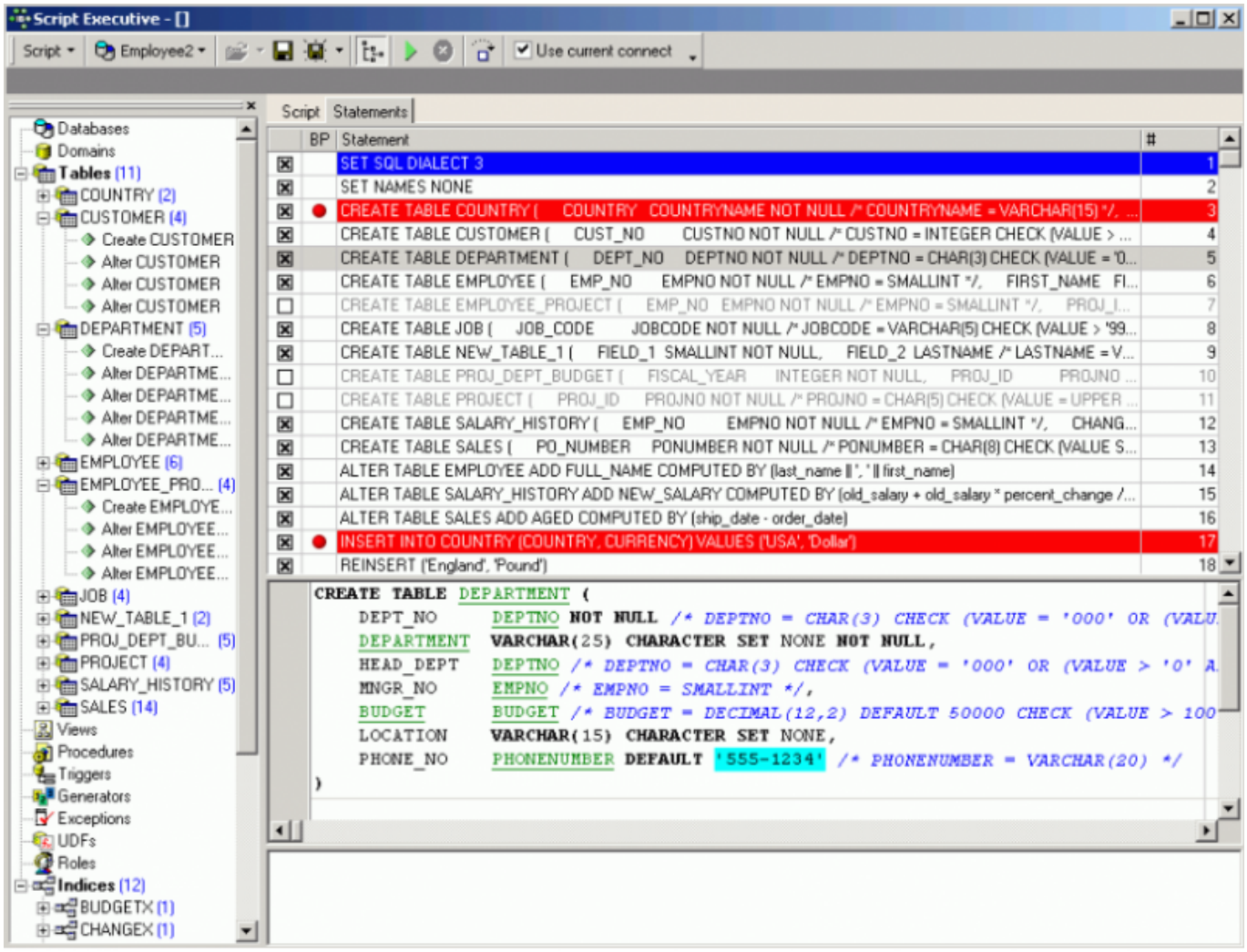
Following statement execution, the Script page displays any errors highlighted in red. Any error messages which may appear on the *Messages* page may be consulted in the Firebird 2.1 error codes documentation. Using the

icon, the script can be executed step by step.

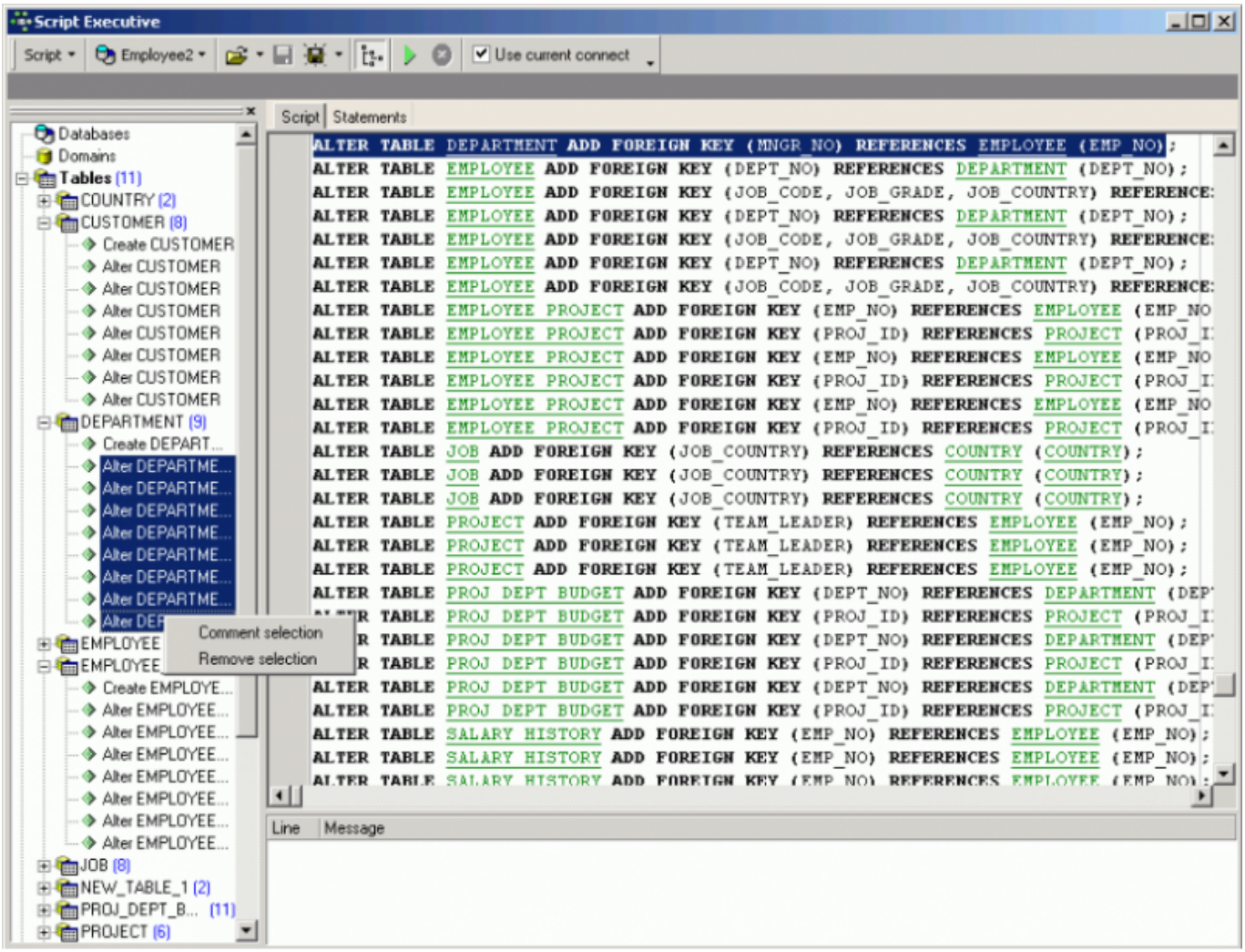
Any errors appearing in the lower *Messages* box may be saved to file if wished, using the right-click menu item *Save Messages Log* ...

Statements page

The *Statements* page displays a list of individual statements in grid form:

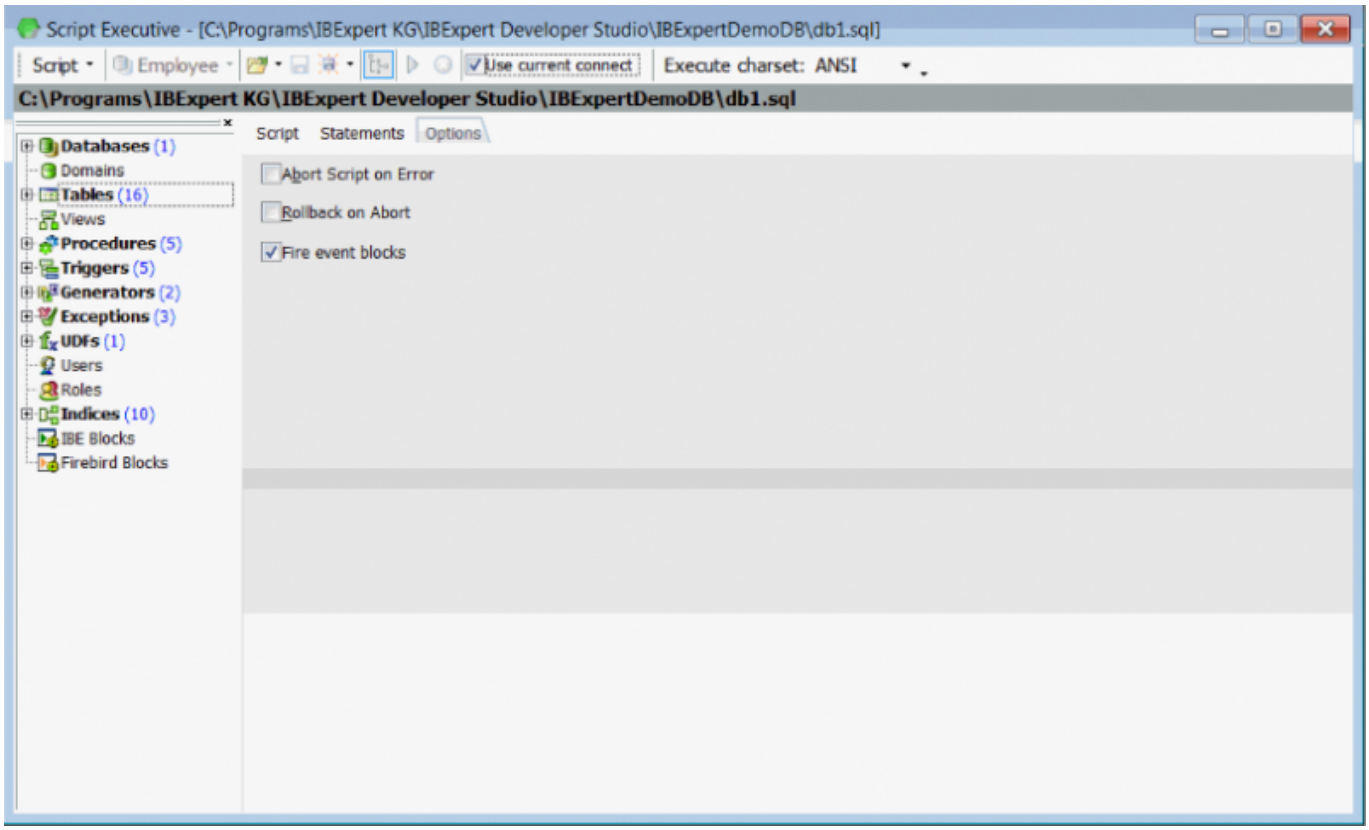


These statements may be removed from the script simply by unchecking the left-hand boxes. One, several or all statements may be checked or unchecked using the right-click menu. Breakpoints can be specified or removed simply by clicking (or using the space bar) to the left of the selected statement in the *BP* column.



back to top of page

Options page



Currently the following options are available which apply to the current instance of the Script Executive:

- **Abort script on error**
- **Rollback on abort**
- **Fire event blocks:** this affects only the current instance of the Script Executive.

back to top of page

Using Unicode in the Script Executive

If you need to insert Unicode text strings into your database, you need to knowm what character set is used for the connection. If a non-unicode charset is used you should not convert your strings to unicode as the server will do this. For example:

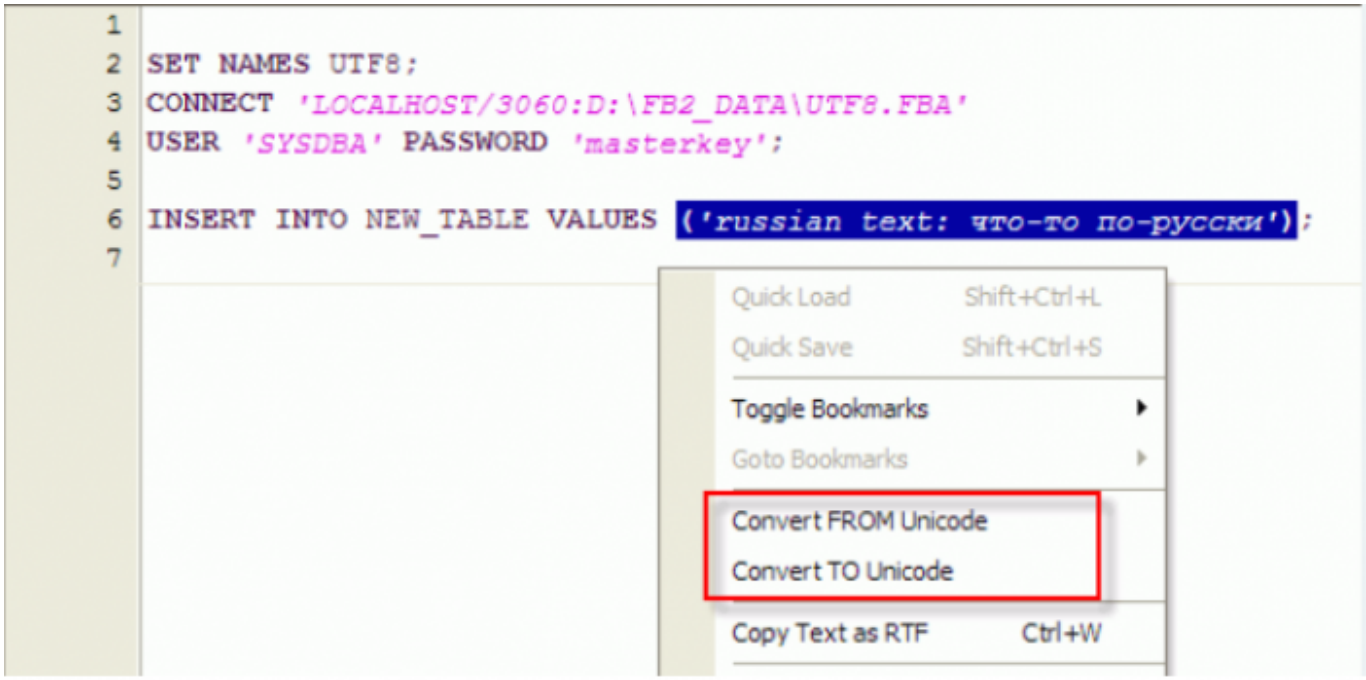
```
SET NAMES WIN1251;
CONNECT ...;

INSERT ... VALUES ('russian text: что-то по-русски');
```

But if you connect to your database with a UTF8 charset you should use unicode strings instead:

```
1  SET NAMES UTF8;
2  CONNECT 'LOCALHOST/3060:D:\FB2_DATA\UTF8.FBA'
3  USER 'SYSDBA' PASSWORD 'masterkey';
4
5  INSERT INTO NEW TABLE VALUES ('russian text: С‡С,Ps-С,Ps РїPs-СЂСѓСЃСЃРєРё');
6
```

To convert strings from/to unicode use the corresponding items in the code editor's popup menu:



back to top of page

Executing multiple scripts from a single script

Simply use the following syntax:

```
connect 'server:c:\my_db.gdb' ...;

input 'c:\my_scripts\f2.sql';
input 'c:\my_scripts\f1.sql';
input 'c:\my_scripts\f3.sql';
```

back to top of page

Create multiple CSV files from a script

The following is an example illustrating the creation of multiple csv files from a script:

```
shell del C:\list.dat nowait;    --deleting the old file
shell del C:\*.csv nowait;    --deleting the old csv files

connect 'localhost:C:\employee.fdb' user 'SYSDBA' password 'masterke';
--connect to employee example database
```

```
output 'C:\list.dat';    --record the following result as a simple text
file, based on each unique employee,
we create a new output ...;select ... ;output; line in the dat file

SELECT distinct
'OUTPUT C:\'||EMPLOYEE.last_name||'.csv delimiter '';'';'||
'SELECT distinct EMPLOYEE.last_name, customer.customer,customer.phone_no '||
'FROM SALES INNER JOIN CUSTOMER ON (SALES.CUST_NO = CUSTOMER.CUST_NO) '||
'INNER JOIN EMPLOYEE ON (SALES.SALES_REP = EMPLOYEE.EMP_NO) where
EMPLOYEE.last_name=XXXXXX||EMPLOYEE.last_name||''';'||
'OUTPUT;'
FROM SALES INNER JOIN CUSTOMER ON (SALES.CUST_NO = CUSTOMER.CUST_NO) INNER
JOIN EMPLOYEE ON

(SALES.SALES_REP = EMPLOYEE.EMP_NO);

output;      --close the dat file
input 'C:\list.dat';  --execute them
```

The data file is created automatically.

The outer query gets one record for each employee, in the inner select, all phone numbers for the employees if customers are selected.

Please also refer to *IBEBlock examples*: Importing data from a CSV file.

back to top of page

Script Language Extensions

Script language extensions are unique to IBExpert, and offer the developer a number of additional language options. These include, among others, conditional directives, DESCRIBE database objects, as well as SET, SHELL, INSERTEX, OUTPUT and RECONNECT.

Conditional Directives

Conditional directives control conditional execution of parts of the script. Four types of conditional directives are supported:

- $IFEXISTS
- $IFIBEVERSION
- $IFNOTEXISTS ($IFNEXISTS)
- $ELSE
- $ENDIF

$IFEXISTS

This tests the existence of the specified database object or data and executes the following block of the script if the object or data do exist in the database.

Syntax

```
1. {$IFEXISTS DOMAIN|TABLE|VIEW|TRIGGER|PROCEDURE|
      EXCEPTION|GENERATOR|UDF|ROLE object_name}
2. {$IFEXISTS select_statement}
```

Example

The following script drops the exception InvalidUserID if it exists in the database:

```
{$IFEXISTS EXCEPTION "InvalidUserID"}

   DROP EXCEPTION "InvalidUserID";
```

The next script alters a procedure:

```
{$IFEXISTS SELECT RDB$PROCEDURE_NAME
           FROM RDB$PROCEDURES
           WHERE RDB$PROCEDURE_NAME = 'GETDBVER'}

   ALTER PROCEDURE GETDBVER
   RETURNS (
       VER INTEGER)
   AS
   begin
     ver = 2;
     suspend;
   end;
```

back to top of page

$IFIBEVERSION

The $IfIBEVersion conditional directive allows you to check the current version of IBExpert/IBEScript.

Syntax

```
        {$IfIBEVersion <relational_operator> <version_number>}
      ...
      ...                 <relational_operator> = < | > | =< | >= | = | <> |
<version_number> - version number string without quote char.
```

Example

```
   {$IfIBEVersion < 2007.7.16.0}
     execute ibeblock
     as
     begin
       ibec_ShowMessage('Please, update your version of
IBExpert/IBEScript!');
     end;
     quit;
```

back to top of page

$IFNOTEXISTS ($IFNEXISTS)

This tests the existence of the specified database object or data and executes the following block of the script if the object or data does not exist in the database.

Syntax

```
1. {$IFNOTEXISTS DOMAIN|TABLE|VIEW|TRIGGER|PROCEDURE|
      EXCEPTION|GENERATOR|UDF|ROLE object_name}
2. {$IFNOTEXISTS select_statement}
```

Example

The following script creates a table CUSTOMERS if there is no such table in the database:

```
{$IFNOTEXISTS TABLE CUSTOMERS}

   CREATE TABLE CUSTOMERS (
     ID          INTEGER NOT NULL PRIMARY KEY,
     FIRST_NAME  VARCHAR(30),
     MIDDLE_NAME VARCHAR(30),
     LAST_NAME   VARCHAR(30));
```

The next script creates an exception:

```
{$IFNOTEXISTS SELECT RDB$EXCEPTION_NAME
              FROM RDB$EXCEPTIONS
              WHERE RDB$EXCEPTION_NAME = 'InvalidUserID'}

   CREATE EXCEPTION "InvalidUserID" 'Invalid User Identifier!';
```

back to top of page

$ELSE

Switches between executing and ignoring the script part are delimited by the previous {$IFEXISTS} or {$IFNOTEXISTS} and the next {$ENDIF}.

Syntax

```
{$ELSE}
```

Example

The following script tests the existence of domain DOM_BOOL in the database. If domain DOM_BOOL cannot be found in the database it will be created. If domain DOM_BOOL already exists in the database it will be altered.

```
{$IFEXISTS DOMAIN DOM_BOOL}

  ALTER DOMAIN DOM_BOOL
  ADD CHECK (VALUE IN (0,1));

{$ELSE}

  CREATE DOMAIN DOM_BOOL AS SMALLINT
  DEFAULT 0 CHECK (VALUE IN (0,1));

{$ENDIF}
```

back to top of page

$ENDIF

Ends the conditional execution initiated by the last {$IFEXISTS} or {$IFNOTEXISTS} directive.

Syntax

```
{$ENDIF}
```

Example

The following script creates a generator:

```
{$IFNOTEXISTS GENERATOR "GenUserID"}

   CREATE GENERATOR "GenUserID";

{$ENDIF}
```

back to top of page

Conditional Directives - the complete example

This example illustrates the use of conditional directives for upgrading databases. Let's assume there is an initial version of your database (version 1):

```
CREATE TABLE FIRST_TABLE (
ID    INTEGER NOT NULL,
DATA  VARCHAR(100));

 CREATE PROCEDURE GETDBVER
 RETURNS (
   VER INTEGER)
 AS
 begin
   ver = 1;
   suspend;
 end;
```

The next script will upgrade a database of any version < 4 to version 4.

```
/***** Upgrade to version 2 *****/
{$IfNotExists select ver from GetDBVer where ver > 1}

ALTER TABLE FIRST_TABLE
ADD CONSTRAINT PK_FIRST_TABLE
PRIMARY KEY (ID);

ALTER PROCEDURE GETDBVER
RETURNS (
  VER INTEGER)
AS
begin
  ver = 2;
  suspend;
end;

{$endif}
```

```
/***** Upgrade to version 3 *****/
{$IfNotExists select ver from GetDBVer where ver > 2}
CREATE GENERATOR GEN_FIRST_TABLE_ID;

CREATE TRIGGER FIRST_TABLE_BI0 FOR FIRST_TABLE
ACTIVE BEFORE INSERT POSITION 0
AS
begin
  new.id = gen_id(gen_first_table_id, 1);
end;

ALTER PROCEDURE GETDBVER
RETURNS (
    VER INTEGER)
AS
begin
  ver = 3;
  suspend;
end;
{$endif}
```

```
/***** Upgrade to version 4 *****/
{$IfNotExists select ver from GetDBVer where ver > 3}
CREATE EXCEPTION DELETION_NOT_ALLOWED 'You cannot delete records!';

CREATE TRIGGER FIRST_TABLE_BD0 FOR FIRST_TABLE
ACTIVE BEFORE DELETE POSITION 0
AS
begin
  exception deletion_not_allowed;
end;

ALTER PROCEDURE GETDBVER
RETURNS (
    VER INTEGER)
AS
begin
  ver = 4;
  suspend;
end;
{$endif}
```

back to top of page

IBExpert DESCRIBE statement

The IBExpert-specific DESCRIBE statement will be translated to the Firebird COMMENT ON statement instead of UPDATE RDB$xxx when executing scripts against Firebird 3 databases.

DESCRIBE DOMAIN

This changes a domain description.

Syntax

DESCRIBE DOMAIN domain_name 'description';

Argument	Description
domain_name	Name of an existing domain.
'description'	Quoted string containing a domain description.

Description

DESCRIBE DOMAIN changes the description of an existing domain domain_name. When the IBExpert Script Executive executes this statement it modifies the value of the RDB$DESCRIPTION column in DB$FIELDS connected with the specified domain name.

Actually the following statement is executed:

```
UPDATE RDB$FIELDS
SET RDB$DESCRIPTION = :DESC
WHERE RDB$FIELD_NAME = 'domain_name'
```

where DESC parameter is filled with the description.

Example

```
DESCRIBE DOMAIN DOM_BOOL
'Boolean value:
 0 - FALSE
 1 - TRUE';
```

back to top of page

DESCRIBE EXCEPTION

This changes an exception's description.

Syntax

```
DESCRIBE EXCEPTION exception_name 'description';
```

Argument	Description
exception_name	Name of an existing exception.
'description'	Quoted string containing a new description of specified exception.

Description

DESCRIBE EXCEPTION changes the description of an existing exception exception_name. When the IBExpert Script Executive executes this statement it modifies the value of the RDB$DESCRIPTION column in RDB$EXCEPTIONS connected with the specified exception. Actually the following statement is executed:

```
UPDATE RDB$EXCEPTIONS
SET RDB$DESCRIPTION = :DESC
WHERE RDB$EXCEPTION_NAME = 'exception_name'
```

where the DESC parameter is filled with the description.

Example

```
DESCRIBE EXCEPTION MISSING_USER
'There is no such user!';
```

back to top of page

DESCRIBE FIELD

This changes a column description.

Syntax

```
DESCRIBE FIELD column_name TABLE table_name 'description';
```

Argument	Description
column_name	Name of an existing column of table table_name.
table	Name of an existing table.
'description'	Quoted string containing a column description.

Description

DESCRIBE FIELD changes the description of an existing column column_name of table table_name. When the IBExpert Script Executive executes this statement it modifies the value of the RDB$DESCRIPTION column in RDB$RELATION_FIELDS connected with the specified column and table names. Actually the following statement is executed:

```
UPDATE RDB$RELATION_FIELDS
SET RDB$DESCRIPTION = :DESC
WHERE (RDB$RELATION_NAME = 'table_name') AND
      (RDB$FIELD_NAME = 'column_name')
```

where the DESC parameter is filled with the description.

Example

```
DESCRIBE FIELD FULL_USER_NAME TABLE USERS
'Full user name.
```

```
 Computed, concatenation of FIRST_NAME, MIDDLE_NAME and LAST_NAME';
```

back to top of page

DESCRIBE FUNCTION

This changes an UDF description.

Syntax

```
DESCRIBE FUNCTION function_name 'description';
```

Argument	Description
function_name	Name of an existing user-defined function.
'description'	Quoted string containing an UDF description.

Description

DESCRIBE FUNCTION changes the description of an existing user-defined function function_name. When the IBExpert Script Executive executes this statement it modifies the value of the RDB$DESCRIPTION column in RDB$FUNCTIONS connected with the specified function. Actually the following statement is executed:

```
UPDATE RDB$FUNCTIONS
SET RDB$DESCRIPTION = :DESC
WHERE RDB$FUNCTION_NAME = 'function_name'
```

where the DESC parameter is filled with the description.

Example

```
DESCRIBE FUNCTION COMPARE_BLOBS
'Compares two blob values and returns 1
if both values are equal. In other case returns 0';
```

back to top of page

DESCRIBE PARAMETER

This changes a procedure parameter description.

Syntax

```
DESCRIBE PARAMETER parameter_name PROCEDURE procedure_name 'description';
```

Argument	Description
parameter_name	Name of an existing parameter of stored procedure.

Argument	Description
procedure_name	Name of an existing stored procedure.
'description'	Quoted string containing a parameter description.

Description

DESCRIBE PARAMETER changes the description of an existing parameter parameter_name of a specified stored procedure procedure_name. When the IBExpert Script Executive executes this statement it modifies the value of the RDB$DESCRIPTION column in RDB$PROCEDURE_PARAMETERS connected with the specified parameter and procedure names. Actually the following statement is executed:

```
UPDATE RDB$PROCEDURE_PARAMETERS
SET RDB$DESCRIPTION = :DESC
WHERE (RDB$PROCEDURE_NAME = 'procedure_name') AND
      (RDB$PARAMETER_NAME = 'parameter_name')
```

where the DESC parameter is filled with the description.

Example

```
DESCRIBE PARAMETER USER_ID PROCEDURE CALC_TRAFFIC
'User ID';
```

back to top of page

DESCRIBE PROCEDURE

This changes a stored procedure description.

Syntax

```
DESCRIBE PROCEDURE procedure_name 'description';
```

Argument	Description
procedure_name	Name of an existing stored procedure.
'description'	Quoted string containing a procedure description.

Description

DESCRIBE PROCEDURE changes the description of an existing stored procedure procedure_name. When the IBExpert Script Executive executes this statement it modifies the value of the RDB$DESCRIPTION column in RDB$PROCEDURES connected with the specified procedure. Actually the following statement is executed:

```
UPDATE RDB$PROCEDURES
SET RDB$DESCRIPTION = :DESC
WHERE RDB$PROCEDURE_NAME = 'procedure_name'
```

where the DESC parameter is filled with the description.

Example

```
DESCRIBE PROCEDURE CALC_TRAFFIC
'Calculates the summary traffic';
```

back to top of page

DESCRIBE TABLE

This changes a table description

Syntax

```
DESCRIBE TABLE table_name 'description';
```

Argument	Description
table_name	Name of an existing table.
'description'	Quoted string containing a table description.

Description

DESCRIBE TABLE changes the description of an existing table table_name. When the IBExpert Script Executive executes this statement it modifies the value of the RDB$DESCRIPTION column in RDB$RELATIONS connected with the specified table. Actually following statement is executed:

```
UPDATE RDB$RELATIONS
SET RDB$DESCRIPTION = :DESC
WHERE RDB$RELATION_NAME = 'table_name'
```

where the DESC parameter is filled with the description.

Example

```
DESCRIBE TABLE CUSTOMERS
'Customers of our excellent application';
```

back to top of page

DESCRIBE TRIGGER

This changes a trigger description

Syntax

```
DESCRIBE TRIGGER trigger_name 'description';
```

Argument	Description
trigger_name	Name of an existing trigger.
'description'	Quoted string containing a trigger description.

Description

DESCRIBE TRIGGER changes the description of an existing trigger trigger_name. When the IBExpert Script Executive executes this statement it modifies the value of the RDB$DESCRIPTION column of RDB$TRIGGERS connected with the specified table. Actually the following statement is executed:

```
UPDATE RDB$TRIGGERS
SET RDB$DESCRIPTION = :DESC
WHERE RDB$TRIGGER_NAME = 'trigger_name'
```

where the DESC parameter is filled with the description.

Example

```
DESCRIBE TRIGGER USERS_BI
'Generates an unique identifier';
```

back to top of page

DESCRIBE VIEW

This changes a view description

Syntax

```
DESCRIBE VIEW view_name 'description';
```

Argument	Description
view_name	Name of an existing view.
'description'	Quoted string containing a view description.

Description

DESCRIBE VIEW changes the description of an existing view view_name. When the IBExpert Script Executive executes this statement it modifies the value of the RDB$DESCRIPTION column of RDB$RELATIONS connected with the specified view. Actually the following statement is executed:

```
UPDATE RDB$RELATIONS
SET RDB$DESCRIPTION = :DESC
WHERE RDB$RELATION_NAME = 'view_name'
```

where the DESC parameter is filled with the description.

Example

```
DESCRIBE VIEW ALL_USERS
```

```
'Just all users...:)';
```

back to top of page

INSERTEX (CSV file import)

This imports data from a CSV-file into a database table.

Syntax

```
INSERTEX INTO table_name [(columns_list)]
   FROM CSV file_name
   [SKIP n]
   [DELIMITER delimiter_char]
```

Argument	Description
table_name	Name of a table into which to insert data.
columns_list	List of columns into which to insert data.
file_name	Name of CSV-file from which to import data.
SKIP n	Allows the first n lines of CSV-file to be skipped while importing data.
DELIMITER delimiter_char	Allows a delimiter to be specified, which will be used for parsing data values.

If this argument isn't specified IBExpert will use a colon as a delimiter.

Description

INSERTEX imports data from a CSV-file into a database table. Values within the CSV-file must be separated with a colon CHAR or any other char. In the latter case it is necessary to specify a delimiter CHAR using the DELIMITER argument. It is also possible to specify non-print characters as a delimiter. For example, if values are separated with tab char (ASCII value $09) it may be specified as DELIMITER #9 or DELIMITER $9.

To ignore unwanted quotes use the QUOTECHAR '"' option.

If a table table_name is missing in the database, it will be created automatically. In this case the number of columns in the newly created table will be equal to the number of values in the first line of the CSV-file. Columns will be named F_1, F_2 etc. The data type of each column is VARCHAR(255).

If the columns_list isn't specified IBExpert will insert data from the very first column. Otherwise data will only be inserted into specified columns. It is possible to skip the first several lines of the CSV-file using the SKIP argument. This may be useful if the first line contains column captions or is empty.

It is also possible to use the INSERTEX command in the SQL Editor.

Examples

Let's consider the use of INSERTEX in the following examples. Assume there is a CSV-file with the following data, delimited with a colon:

```
C:\Mydata.csv
=====================================================
ID:FIRST_NAME:LAST_NAME:SEX
1:John:Doe:M
2:Bill:Gates:M
3:Sharon:Stone:F
4:Stephen:King:M
=====================================================
```

The following INSERTEX statement creates a table PEOPLE (if it doesn't already exist) and fills it with data from C:\Mydata.csv:

```
INSERTEX INTO PEOPLE FROM CSV 'C:\Mydata.csv' DELIMITER ':';
```

The structure and contents of PEOPLE after the data import are shown below:

F_1 (VARCHAR(255))	F_2 (VARCHAR(255))	F_3 (VARCHAR(255))	F_4 (VARCHAR(255))
ID	FIRST_NAME	LAST_NAME	SEX
1	John	Doe	M
2	Bill	Gates	M
3	Sharon	Stone	F
4	Stephen	King	M

The following INSERTEX statement is almost identical to the one above, but here the first line of the CSV-file has been skipped:

```
INSERTEX INTO PEOPLE FROM CSV 'C:\Mydata.csv' DELIMITER ':' SKIP 1;
```

The structure and content of the PEOPLE table after import is shown below:

F_1 (VARCHAR(255))	F_2 (VARCHAR(255))	F_3 (VARCHAR(255))	F_4 (VARCHAR(255))
1	John	Doe	M
2	Bill	Gates	M
3	Sharon	Stone	F
4	Stephen	King	M

In the next example the PEOPLE table is created first, and then subsequently populated with the data from C:\Mydata.csv:

```
CREATE TABLE PEOPLE (
    ID         INTEGER NOT NULL,
    FIRST_NAME VARCHAR(30),
    LAST_NAME  VARCHAR(30),
    SEX        CHAR(1));

INSERTEX INTO PEOPLE FROM CSV 'C:\Mydata.csv' DELIMITER ':' SKIP 1;
```

Below the structure and content of the PEOPLE table after import:

ID (INTEGER)	FIRST_NAME (VARCHAR(30))	LAST_NAME (VARCHAR(30))	SEX (CHAR(1))
1	John	Doe	M
2	Bill	Gates	M
3	Sharon	Stone	F
4	Stephen	King	M

In the next example only three columns (ID, FIRST_NAME and LAST_NAME) are affected:

```
CREATE TABLE PEOPLE (
    ID         INTEGER NOT NULL,
    FIRST_NAME VARCHAR(30),
    LAST_NAME  VARCHAR(30),
    SEX        CHAR(1));

INSERTEX INTO PEOPLE (ID, FIRST_NAME, LAST_NAME)
    FROM CSV 'C:\Mydata.csv'
    DELIMITER ':' SKIP 1;
```

The structure and content of the PEOPLE table after import can be seen below:

ID (INTEGER)	FIRST_NAME (VARCHAR(30))	LAST_NAME (VARCHAR(30))	SEX (CHAR(1))
1	John	Doe	NULL
2	Bill	Gates	NULL
3	Sharon	Stone	NULL
4	Stephen	King	NULL

back to top of page

OUTPUT

This redirects the output of SELECT statements to a named file.

Syntax

```
OUTPUT [filename [DELIMITER delim_char]
                 [QUOTECHAR 'quote_char']
                 [TIMEFORMAT 'time_format']
                 [DATEFORMAT 'date_format']
                 [DECIMALSEPARATOR 'dec_sep']
                 [NULLS]
                 [FIELDNAMES]
                 [ASINSERT | AsUpdateOrInsert [INTO table]]
```

Argument	Description
filename	Name of the file in which to save output.

Argument	Description
DELIMITER delim_char	Determines a delimiter character which is used for separating field values. If the delimiter is not specified, or the empty string is specified as a delimiter, outswapping of the data will be carried out in the format with the fixed positions of fields. It is also possible to specify a delimiter character as a decimal or hexadecimal value of the character code. For example, to set the tab character (ASCII value $09) as a delimiter, simply specify DELIMITER #9 or DELIMITER $9.
QUOTECHAR 'quote_char'	Defines the character which will be used for quoting string values. If this argument is not specified or an empty string is specified, string values will not be quoted.
TIMEFORMAT 'time_format'	Defines the string which will be used for formatting the values of time fields and the time slice of datetime values. If the argument is not defined, time values will be unloaded in the native InterBase® format (for example, 17:15:45).
DATEFORMAT 'date_format'	Defines the string which will be used for formatting values of date fields and the date part of datetime values. If the argument is not defined, date values will be unloaded in the native InterBase® format (for example, 17-FEB-2001).
DECIMALSEPARATOR 'dec_sep'	Defines the decimal separator which is used when outswapping the data. If this argument is not defined, the system decimal separator is used.
NULLS	Defines how NULL values will be output. If the argument is not specified, NULLs are output as an empty string. Otherwise NULLs will be unloaded as the string <null>.
FIELDNAMES	If this argument is specified, the first line in the resulting file will be a line with names of SELECT columns.
ASINSERT	This argument allows data to be unloaded as a set of INSERT operators, i.e. to get a usual SQL script.
INTO table	It is used together with ASINSERT for redefining the name of the table in INSERT operators. If the argument is not given, the name of the first table in the record set will be used.
AsUpdateOrInsert	Produces a script containing UPDATE OR INSERT statement.
OctetsAsHex	Allows the extraction of CHAR(n) CHARACTER SET OCTETS values in hexadecimal format.
MemoAsText	Implemented in IBExpert version 2020.09.13, if this option is specified blob values of subtype 1 (TEXT) will be exported as strings.

Description

The OUTPUT operator is intended for redirecting the output of SELECT statements in an external file. With the help of the given operator it is possible to export the data easily into a file with separators or with a fixed column position. OUTPUT without parameters closes the file which was opened with the previous OUTPUT command, and resets all export customizations to default.

If ASINSERT is not specified, blob fields are ignored when outswapping the data. Using ASINSERT even blob values are exported, i.e. an additional file with the extension .lob is created, in which all blob fields are stored.

While outputting into SQL script (ASINSERT is specified) DELIMITER, QUOTECHAR, NULLS and FIELDNAMES arguments are ignored.

Examples

The following script creates a MyData.txt file in the current directory and outputs the data of the SELECT into it, with a fixed column position format. If MyData.txt file already exists in the current directory, the data will be appended to it.

```
OUTPUT MyData.txt;
SELECT * FROM MY_TABLE;
OUTPUT;
```

In the next example the data will be exported in the comma-separated values (CSV) format:

```
OUTPUT 'C:\MyData\MyData.csv' DELIMITER ';'
                              FIELDNAMES
                              QUOTECHAR '"'
                              DECIMALSEPARATOR '.';
SELECT * FROM MY_TABLE;
OUTPUT;
```

In the following script the data will be exported into SQL script as a set of INSERT operators:

```
OUTPUT 'C:\MyScripts\Data.sql' ASINSERT INTO "MyTable";
SELECT * FROM MY_TABLE;
OUTPUT;
```

The next example illustrates usage of the OUTPUT statement together with SHELL.

```
/* First create a folder C:\MyData*/
SHELL MKDIR C:\MyData;

/* Try to delete mydata.csv */
SHELL DEL C:\MyData\mydata.csv;

/* Redirect output of SELECTs into mydata.csv */
OUTPUT C:\MyData\mydata.csv DELIMITER ';'
                            DATEFORMAT 'MMMM-dd-yyyy'
                            TIMEFORMAT 'hh:nn:ss.zzz'
                            QUOTECHAR '"';

SELECT * FROM MY_TABLE;

/* Close C:\MyData/mydata.csv */
OUTPUT;

/* Try to open just created CSV-file with Windows Notepad */
SHELL notepad.exe C:\MyData\mydata.csv NOWAIT;

/* Try to open C:\MyData\mydata.csv with the application
   associated with CSV files */
SHELL C:\MyData\mydata.csv NOWAIT;
```

Example using the AsUpdateOrInsert option:

```
OUTPUT 'C:\MyScripts\data.sql' ASUPDATEORINSERT;
SELECT * FROM MYTABLE ORDER BY ID;
OUTPUT;
COMMIT;
```

Example using the OctetsAsHex option:

```
OUTPUT 'D:\MyData\data.sql' AS INSERT OctetsAsHex;
SELECT * FROM MYTABLE
OUTPUT;
```

Extended syntax of OUTPUT command:

1.

```
output 'E:\data.sql'
as insert into mytable commit after 1000;
select * from IBE$$TEST_DATA where F_INTEGER < 3000;
output;
```

2.

```
output 'E:\data.sql'
as reinsert into mytable
commit after 2000;
select * from IBE$$TEST_DATA where F_INTEGER < 3000;
output;
```

3.

```
output 'E:\data.sql'
as execute procedure myproc;
select * from IBE$$TEST_DATA where F_INTEGER < 3000;
output;
```

The ASINSERT option is available for compatibility.

back to top of page

RECONNECT

RECONNECT closes the current connection and creates a new one with the same parameters (database, user name, password etc.).

Syntax

```
RECONNECT;
```

back to top of page

REINSERT

IBExpert has introduced the new REINSERT statement. Directly following an INSERT it is possible to perform further INSERTs with new contents.

back to top of page

SET BLOBFILE

SET BLOBFILE is a special extension of script language that allows IBExpert's Script Executive to execute scripts containing references to blob field values.

IBExpert uses an original mechanism to extract values of blob fields into a script. This allows you to store the entire database (metadata and data) into script files and execute these scripts with IBExpert. A small example illustrates the method used to extract blob values.

For example, your database has a table named COMMENTS:

```
CREATE TABLE COMMENTS (
   COMMENT_ID INTEGER NOT NULL PRIMARY KEY,
   COMMENT_TEXT BLOB SUBTYPE TEXT);
```

This table has three records:

COMMENT_ID	COMMENT_TEXT
1	First comment
2	NULL
3	Another comment

If the *Extract BLOBs* option is not checked, you will receive the following script:

```
CREATE TABLE COMMENTS (
   COMMENT_ID INTEGER NOT NULL PRIMARY KEY,
   COMMENT_TEXT BLOB SUBTYPE TEXT);

INSERT INTO COMMENTS (COMMENT_ID) VALUES (1);
INSERT INTO COMMENTS (COMMENT_ID) VALUES (2);
INSERT INTO COMMENTS (COMMENT_ID) VALUES (3);
```

... and, of course, you will lose your comments if you restore your database from this script.

But if the *Extract BLOBs* option is checked IBExpert will generate quite a different script:

```
SET BLOBFILE 'C:\MY_SCRIPTS\RESULT.LOB';
```

```
CREATE TABLE COMMENTS (
   COMMENT_ID INTEGER NOT NULL PRIMARY KEY,
   COMMENT_TEXT BLOB SUBTYPE TEXT);

INSERT INTO COMMENTS (COMMENT_ID, COMMENT_TEXT) VALUES (1,
h0000000_0000000D);
INSERT INTO COMMENTS (COMMENT_ID, COMMENT_TEXT) VALUES (2, NULL);
INSERT INTO COMMENTS (COMMENT_ID, COMMENT_TEXT) VALUES (3,
h000000D_0000000F);
```

Also IBExpert generates a special file with the extension .lob where blob values are stored. In the current example result.lob will be 28 bytes long and its contents will be the first commentAnother comment.

back to top of page

SET CLIENTLIB

This defines the client library to be used while executing a script.

Syntax

```
SET CLIENTLIB file_name;
```

Argument	Description
file_name	Client library file name.

Description

SET CLIENTLIB defines client library which will be used while executing a script. The default client library is gds32.dll.

Example

```
SET CLIENTLIB 'C:\Program Files\Firebird\Bin\fbclient.dll';
```

back to top of page

SET LOGIN PROMPT

This command enables or disables the login prompt dialog when the user name and/or password within CONNECT or CREATE DATABASE is not specified. The SET LOGIN PROMPT command is useful when you use trusted authentication and don't need a login prompt dialog. The default value is ON.

Example

```
SET NAMES UTF8;
SET LOGIN PROMPT OFF;
CONNECT ...;
```

```
...
```

back to top of page

SET PARAMFILE

PARAM file is an ini-file with param values.

For example, if your script contains some parameterized INSERT/UPDATE/DELETE statements you can define parameter values in an external file (params file):

```
param1=12-FEB-2003
param2=John Doe
param3=35
...
```

When IBEScript finds a query with parameters it looks for the values of these parameters in the specified params file.

back to top of page

SET TRPARAMS

The SET TRPARAMS command allows you to specify your own parameters of the script transaction instead of default ones.

Syntax

```
   SET TRPARAMS '<params>';
   where <params> is a list of transaction parameters separated by commas or
   spaces.
        Example:
        SET TRPARAMS 'isc_tpb_concurrency, isc_tpb_nowait';
```

Note: If the current transaction is active SET TRPARAMS will commit it and, following that, change the transaction parameters.

back to top of page

SHELL

This allows execution of an operating system command.

Syntax

```
SHELL os_command [NOWAIT];
```

Argument	Description
os_command	An operating system command.
NOWAIT	Optional argument. If specified, execution of a script will be continued right after creation of the process executing the command of operating system, not waiting its completion.

Description

The SHELL operator tries to execute the command os_command. If NOWAIT is not specified, the further execution of a script stops before completion of the process created by SHELL operator. Otherwise script execution will be continued immediately after beginning the execution of the command os_command.

Examples

The following script tries to create a folder MyFolder in the current directory:

```
SHELL mkdir MyFolder;
```

The following example shows the use of the SHELL command to start Notepad.exe and the loading of C:\MyTexts\Shedule.txt file in it. It is necessary to use NOWAIT here, otherwise it is not possible to execute the script further, and it will be impossible to resume work in IBExpert until the Notepad is closed.

```
SHELL "notepad.exe C:\MyTexts\Shedule.txt" NOWAIT;
```

The next example illustrates the use of the SHELL statement together with OUTPUT.

```
/* First create a folder C:\MyData*/
SHELL MKDIR C:\MyData;

/* Try to delete mydata.csv */
/>SHELL DEL C:\MyData\mydata.csv;

/* Redirect output of SELECTs into mydata.csv */
OUTPUT C:\MyData\mydata.csv DELIMITER ';'
                           DATEFORMAT 'MMMM-dd-yyyy'
                           TIMEFORMAT 'hh:nn:ss.zzz'
                           QUOTECHAR '"';


SELECT * FROM MY_TABLE;

/* Close C:\MyData\mydata.csv */
OUTPUT;

/* Try to open just created CSV-file with Windows Notepad */
SHELL notepad.exe C:\MyData\mydata.csv NOWAIT;

/* Try to open C:\MyData\mydata.csv with the application
   associated with CSV files */
```

```
SHELL C:\MyData\mydata.csv NOWAIT;
```