NatStar

Version 5.00 Edition 1

NS-DK

Version 5.00 Edition 1

NatWeb

Version 4.00 Edition 1

DB2

Ref. n° NS500DB2U00033

Contents

About this Manual

This is the DB2 manual for Nat System‘s development tools. This manual describes the DB2 interface allowing the access to an DB2 database.

Supported configurations

Development environment

- Windows 32 bits : 95, 98, NT 4.0, 2000

Client environment

Operating system	**DBMS drivers available**
Windows NT4, 98, 95, 2000 32 bits	DB2 2.1, 5.0, 6.1, 7.2
Windows XP 32 bits	DB2 7.2, 8.2

Server environment

Operating system	**DBMS drivers available**
Windows NT Windows 2000 32 bits	DB2 2.1, 5.0, 6.1, 7.2, 8.2
Windows 2003 32 bits	DB2 7.2, 8.2
MVS/CICS/IMS MVS/Batch	DB2

Relationship to other manuals

☞ Before reading this manual you are expected to have read the « Overview » and « Getting started » manuals. You should not need to use this manual unless you have been advised to do so or if you are already an experienced Nat System developer. If this is the case, you can use this manual to learn in detail about the components it describes.

☞ Strictly speaking, in standard use of NatStar’s Information Modeling tool, you don’t have to program data accesses yourself. The Information Modeling engine takes care of that. In this case, you don’t need to look at the libraries described in this manual. However this manual will prove usefeul if you want to program your applications’ data accesses yourself.

Organization of the manual

This manual contains one chapter, which describes the set of API components of the DB2 interface.

Chapter 1	**DB2 interface** Describes the functions of the NSnnDB2x library associated with the DB2 database.

Conventions

Typographic conventions

Important term	Important terms are printed in **bold**.
Interface component	The names of windows, dialog boxes, controls, buttons, menus and options are printed in *italics*.
[F9]	Function key names appear in square brackets.
FILENAME	Filenames are printed in UPPERCASE.
`syntax example`	Syntax examples are printed in a `fixed-width font.`

Notational conventions

•	A round bullet is used for lists
♦	A diamond is used for alternatives
1.	Numbers are used to mark the steps in a procedure to be carried out in sequence

Operating conventions

Choose *XXX \ YYY*	This means you need to open the *XXX* menu, then choose the *YYY* command (option) from this menu. You can perform this action using the mouse or mnemonic characters on the keyboard.
Click the *XXX \ YYY* button	This means you need to display the tool bar named *XXX*, then click the *YYY* button in this tool bar (the name of each button is shown by its help bubble). You can only perform this action with the mouse.
Choose the *XXX* button	This means you need to choose the *XXX* button in a dialog box. You can perform this action using the mouse or mnemonic characters on the keyboard.

Icon codes

☞	**Comment,** note, etc.
👓	**Reference** to another part of the documentation
⚠	**Danger**: precaution to be taken, irreversible action, etc.
💡	**Suggestion**: helpful hints, etc.
⇗	**To go a step further**: level of detail or expertise greater than the average level of the document
⊞	Indicates specific information on using the software under DOS- Windows (all versions)

32	Indicates specific information on using the software under DOS-Windows 32 bits
32 bits	Indicates specific information on using the software under DOS-Windows 32 bits
UNIX	Indicates specific information on using the software under Unix systems

Chapter 1 DB2 interface

The NSnnDB2x library allows your applications built with Nat System development tools to interface with client versions of DB2.

Nat System products support in environment Windows 32 bits : DB2 2.1, 5.0, 6.1 and 8.2.

To ease the writing in the documentation, NSnnDB21, NSnnDB25, NSnnDB26 and NSnnDB28 on NSnnDB2x generic name.

Nat System recommands highly the use of the static stored procedures instead of the stored procedures with DB2 databases.

This chapter explains

- How to install this library
- The components in this library, arranged in functional categories
- The reference of the components in this library
- The reference of the NS_FUNCTION in this library

Contents

Introduction

The NSnnDB2x libraries allows your applications built with NAT SYSTEM to interface with client versions of DB2.

It also supports the RECORD and REEXECUTE commands which combine the adaptability of dynamic SQL with the speed of static SQL. In fact, a dynamic SQL order is used when the RECORD command is executed. When the REEXECUTE command is executed, as the analysis of query has already been done by the motor, only the values of the host variables are set.

Correspondence between drivers and DB2's versions

The name of the driver is used with THINGS_DB_INIT instruction for NatStar and SQL_INIT instruction for NatStar, NatWeb and NS-DK.

The following table presents DB2 versions and the corresponding drivers.

DB2's versions	**Driver**
DB2 2.1	NSnnDB21.dll
DB2 5.0	NSnnDB25.dll
DB2 6.1	NsnnDB26.dll
DB2 8.2	NSnnDB28.dll

☞ nn indicates the target platform:

- w2 for Windows 32 bits
- w4 for Windows 64 bits

⚠ To ease the writing of the documentation NSnnDB21, NSnnDB25 and NSnnDB26 will be gathered in NSnnDB2x.generic name.

Installation

Copy the file NSnnDB2x.DLL into the directory that contains the DLLs for your NAT SYSTEM environment (C:\NATSTAR\BIN, C:\NATWEB\BIN, and so on.)

The SQL libraries supplied with your Nat System development tools interface with the DLLs supplied by the DBMS manufacturer. In some cases, a utility also needs to be run. Check your configuration using the manuals supplied by your DBMS vendor.

NSnnDB2x drivers use CLI level 2.

Implicit Output Data Conversions

Ws strongly suggest reading the DB2x manual and the documentation furnished with your DB2x drivers for more information about conversions.

However, for certain data, use the following conversions:

DB2x	**NCL**
DECIMAL, NUMERIC	CSTRING (*), CHAR, INT, NUM
BINARY, VARBINARY	BLOBS, SEGMENT
TEXT, IMAGE	BLOBS, SEGMENT

(*)We strongly suggest using this type of data.

SQL static generation

Advantages of static SQL

NatStar and NatWeb dynamically generate SQL database access queries. This means that it builds these queries while the application runs, according to the context.

Many database managers, such as Oracle, use dynamically-generated SQL queries.

However, there are some environments where dynamic SQL queries are unavailable or using them isn't standard practice, such as DB2 under MVS and AS400. In these environments, SQL queries are usually generated statically: SQL queries on tables are defined when the application is generated, not while it runs. They are independent of the context they will be executed in.

The advantage of static SQL generation is that it improves an application's performance because:

- It eliminates the phase that builds queries at run time.
- It allows early access optimization by the SQL engine.

For these reasons, NatStar and NatWeb provide a configuration option that lets you generate static SQL instead of dynamic SQL.

How to generate static SQL

In dynamic SQL mode, NatStar and NatWeb generate the C sources associated with your application's classes, then it automatically executes the phases that transform these sources into binary files (compilation, etc.). In static SQL mode, NatStar and NatWeb generates classes in a different way: the application's C sources are obtained after a precompilation phase. The aim of this precompilation phase is to incorporate your SQL queries into your C sources and database. A **preprocessor** supplied with the database manager carries out this precompilation, which means that NatStar and NatWeb can't generate the C sources directly. Therefore, NatStar and NatWeb generation phase is confined to building the files that can be precompiled. After this, you're responsible for running the precompiler and transforming the .C files into binary files.

Because static SQL requires a different generation technique for classes, you need to divide your generation procedure into two phases:

In the first phase, you use NatStar and NatWeb's standard generation procedure to generate the binary files associated with your libraries, window models, process models and domains.

You use a standard generation configuration. Static SQL doesn't impact any of these resources. You'll obtain the application's dynamic libraries (DLLs) and executable.

In the second phase, you use NatStar and NatWeb with your external tools to produce the binary files associated with your classes.

Generate your classes in NatStar and NatWeb using a special configuration for static SQL. This NatStar and NatWeb generation procedure produces special files instead of the application's binary files.

Make sure the database associated with your application physically exists. If it doesn't, you can generate it with NatStar and NatWeb.

Transform the special files built by NatStar and NatWeb into binary files. To do this, run the precompiler, binder, compiler and linker.

☞ You can now specify not to automatically convert static SQL external table and colum names into all caps. To make Static SQL generation case sensitive, add the following to you CONFIG.INI file:

```
[Generator]
StaticSqlCaseSensitive=1
```

Any other value assigned to this setting has no effect and SQL generation continues to into all caps.

Restrictions

When you use static SQL in NatStar and NatWeb, restrictions apply in the following areas:

- Where clauses
- External representations

Moreover, static SQL isn't fully compatible with the previous versions of NatStar and NatWeb.

Cannot code SELECT INTO when doing SQL STATIC. You must use SELECT and FETCH INTO.

Where clauses

You can only statically generate Where clauses that have been defined as keys of a class (named Where clauses). Any Where clauses entered in your LData templates or Iterator templates are described in your libraries and can't be generated in static SQL.

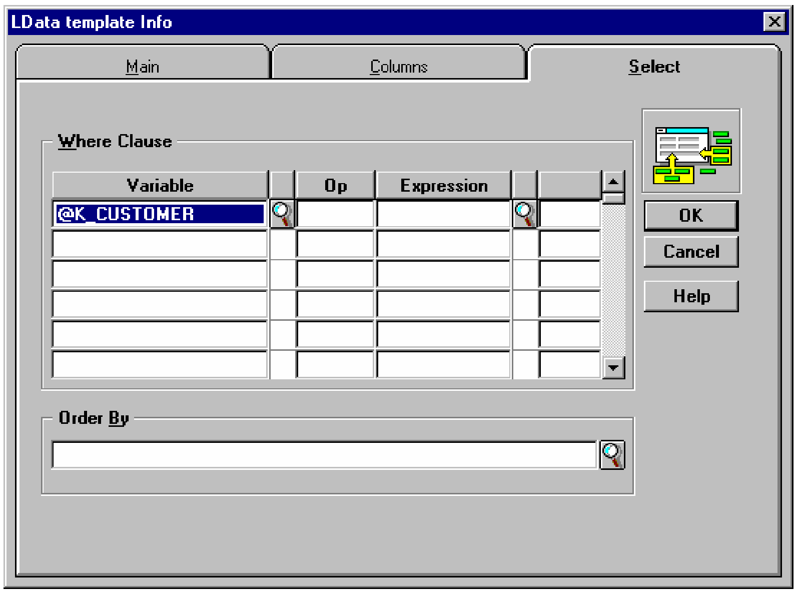
Even if you've requested static SQL generation, NatStar will continue to generate unnamed clauses dynamically.

Using named clauses in an LData template or Iterator template

There is a technique that allows you to use named clauses in an LData template or Iterator template. This technique compensates for the problem of generating clauses in LData templates and Iterator templates using static SQL.

In effect, you can enter the name of a named clause in the *Variable* field of the Where clause defined for the LData template or Iterator template. You simply need to prefix the clause's name with a '@' character to distinguish it from a variable name. The named clause must be defined in the class associated with the LData template or Iterator template.

In this case, the named clause is executed.



Variable	Op	Expression	
@K_CUSTOMER			

In the above example, a named clause is invoked from an LData template's Info box. The syntax '@K_CUSTOMER' indicates that the clause invoked is K_CUSTOMER, which must be defined as a key for the class associated with the LData template.

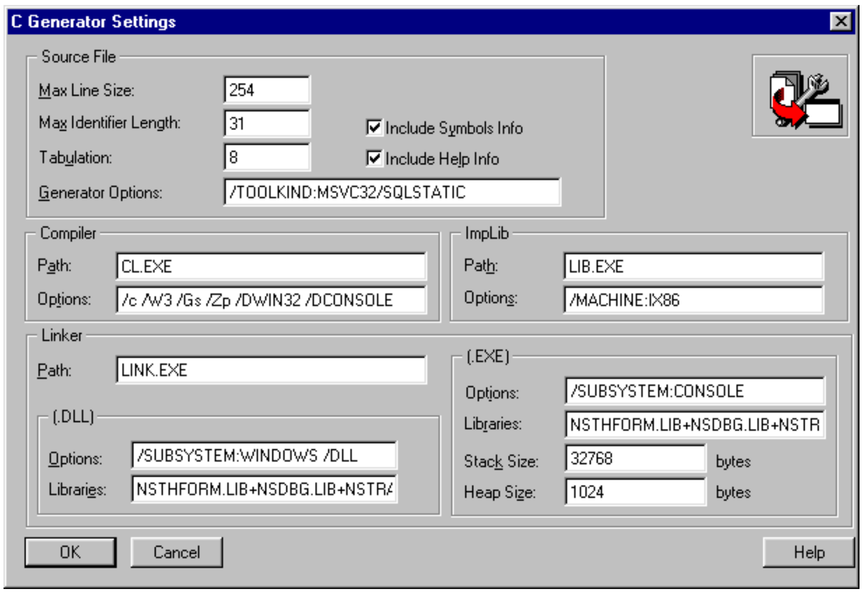
External representations

If you modify a variable's type using external database representations, static SQL ignores this change. If you modify the type of a variable using the database external representations, this may cause a problem when building the binaries (displaying an error message), since there will possibly be a discrepancy between the database column definition, corresponding to an external representation, and the variable as it is defined in the model.

Compiling static requests for DB2

For libraries containing SQL code to generate static SQL, execute the following instructions :

1. In the *Options/Configuration* ... menu, the *Select Default Configuration* box opens. Activate the *Gen...* button. The *C Generator Setting* box opens. In the *Generator Options* field, add /SQLSTATIC.



Source File

Max Line Size:	254
Max Identifier Length:	31
Tabulation:	8
Generator Options:	/TOOLKIND:MSVC32/SQLSTATIC

✓ Include Symbols Info
✓ Include Help Info

Compiler

Path:	CL.EXE
Options:	/c /W3 /Gs /Zp /DWIN32 /DCONSOLE

ImpLib

Path:	LIB.EXE
Options:	/MACHINE:IX86

Linker

Path:	LINK.EXE

(.DLL)

Options:	/SUBSYSTEM:WINDOWS /DLL
Libraries:	NSTHFORM.LIB+NSDBG.LIB+NSTR/

(.EXE)

Options:	/SUBSYSTEM:CONSOLE
Libraries:	NSTHFORM.LIB+NSDBG.LIB+NSTR
Stack Size:	32768 bytes
Heap Size:	1024 bytes

In the *Select Default Configuration* dialog box, don’t click on the *Default* button because it deletes files on %NS-PRIV%\TMP\C_xxx sub-directories.

2. Execute the compilation of librarie(s) containing SQL code to generate in static.

3. Execute the DB2 command interpreter.

4. Connect to the database by typing for example:

```
CONNECT TO sample USER db2user USING db2user;
```

where Sample is the database’s name, User the name of the user and Using his password.

5. Precompile the extension file SQC with the "prep" command. For example

```
PREP %NS-PRIV%\TMP\C_xxx\source.SQC QUALIFIER db2user;
```

6. Access to the *C Generator Setting* dialog box. In *Generator Options*, replace /SQLSTATIC by /NOGEN. Thus, you deactivate the NCL « C » generation and then you don’t crush the C file generated by the precompiler.

☞ Verify the *.DLL* and *.EXE* fields, the *Librairies* field contains the following libraries db2api.lib, db2apie.lib, db2cli.lib supplied with DB2. Software Developer's Kit.

7. Verify that in your environment, the LIB variable referes to the place where the db2api.lib, db2apie.lib, db2cli.lib libraries are and INCLUDE variables to <SQLLIB>\INCLUDE. <SQLLIB> is the place where DB2 is installed in your environment.

8. Recompile to generate the DLL.

Functional categories of NSnnDB2x library

Here is a list, arranged by functional category, of the instructions, functions and constants in the NSnnDB2x library.

Advanced handling of SQL requests

Managing errors

NSnnDB2x library reference

SQL_INIT instruction

Loads the driver needed to use a given version of DB2 for a given target.

Syntax **SQL_INIT** *DLL-name*

Parameters	*DLL-name*	CSTRING	I	name of the driver to load

Notes

1. This must be the first SQL_ instruction called by any application that wants to use DB2x with NCL: it is responsible for loading the library.
2. The *DLL-name* parameter should contain the name of the DLL used to access the DB2 database: NSnnDB2x.
3. The reason why the *DLL-name* parameter has the **NSnnDB2x** format is because there are as many DB2 drivers as targets.

   ☞ nn stands for the version number of the interface that you have installed: w2 for Windows 32 bits, p2 for OS/2 32 bits, 02 for OS/2 16 bits and Windows 16 bits.

Example

```
SQL_INIT "NS02DB26"   ; Loads the DLL which interfaces with
                      ; the client version of DB2 1.2
                      ; on a Windows 3.x (16 bits) machine
...
SQL_STOP              ; Unloads the DLL
```

See also SQL_STOP, SQL_INITMULTIPLE%, SQL_STOPMULTIPLE, SQL_STOPALL, SQL_ERROR%, SQL_ERRMSG$

SQL_STOP instruction

Terminates use of a DBMS by closing all databases and cursors.

Syntax	**SQL_STOP**
Example	See the example of the SQL_INIT instruction.
See also	SQL_INIT, SQL_INITMULTIPLE%, SQL_STOPMULTIPLE, SQL_STOPALL, SQL_ERROR%, SQL_ERRMSG$

SQL_OPEN instruction

Opens a database.

Syntax **SQL_OPEN** *logical-DBname, connection-string*

Parameters	*logical-DBname*	CSTRING	I	logical name of the database to open
	connection-string	CSTRING	I	connection string for a local or remote database

Notes

1. DB2x allows you to open several databases on a network as follows.
2. The *logical-DBname* parameter specifies the logical database name. It can be split into two parts:
   The first part contains a name that identifies the SQL_OPEN statement.

   The second part allows you to connect in different ways.

   - case 1 : **SQL_OPEN "myname" , "scott/tiger@LOCALDB2"**

     This is the standard case for connecting to the database. The second parameter passed to SQL_OPEN must contain all the information required for the connection. The DATA SOURCE must be configured correctly.

   - case 2 : **SQL_OPEN "myname/prompt " , "@LOCALDB2"**

     In this case, the second parameter becomes optional. It initializes the DATA SOURCE and calls the selected driver's connection window.

     ⚠ You need an active window to use this mode.

   - case 3 : **SQL_OPEN "myname/required" , "scott@LOCALDB2"**

     In this last case, if the second parameter is sufficient for connecting to the database, the connection will go ahead. Otherwise, the corresponding driver's connection window will be displayed automatically so that any missing information can be supplied.

     ⚠ You need an active window to use this mode.
3. The connection-string parameter specifies the command string used to connect to a local or remote database, as follows:

`"USERID[/PASSWORD][@<CONNECTOR>]`

where :

`[USERID]`	name of the user account

optional if the data source does not require them or if the DATA SOURCE is correctly configured.

`[PASSWORD]`	password for the user account

optional if the data source does not require them or if the DATA SOURCE is correctly configured.

`[@<CONNECTOR>]`	stands for the name of the DATA SOURCE that you want to access.

If it is not specified, the default data source will be accessed. SQL_EXEC is used to specify which open database will receive the query.

4. If a password is not required to open the database, pass an empty string in *connection-string*.

Example

```
SQL_OPEN "BASE1", "@LOCALDB2"
IF SQL_ERROR% <> 0
      MESSAGE "Error BASE1", SQL_ERRMSG$(SQL_ERROR%)
ENDIF
SQL_OPEN "Base2", "SCOTT/TIGER@NETDB2"
IF SQL_ERROR% <> 0
      MESSAGE "Error BASE2", SQL_ERRMSG$(SQL_ERROR%)
ENDIF
...
SQL_CLOSE "BASE2"
SQL_CLOSE "BASE1"
```

See also SQL_CLOSE, NS_FUNCTION CHANGEDBCNTX, AT, NS_FUNCTION GETDBNAME, SQL_ERROR%, SQL_ERRMSG$

SQL_CLOSE Instruction

Closes a database.

Syntax	**SQL_CLOSE** *logical-DB-name*			
Parameters	*logical-DB-name*	CSTRING	I	logical name of the database to close

Note

1. Although we recommend that you close the databases opened by an application, an SQL_CLOSE instruction is automatically generated for these databases when an application is closed.

Example

See the example of the SQL_OPEN instruction.

See also SQL_OPEN, NS_FUNCTION CHANGEDBCNTX, AT, NS_FUNCTION GETDBNAME, SQL_ERROR%, SQL_ERRMSG$

AT Command

Specifies the name of the logical database affected by the SQL statement that follows.

Syntax **AT** *logical-DBname*, *SQL-statement*

Parameters

logical-DBname	CSTRING	I	logical database name
SQL-statement	CSTRING	I	SQL statement to execute

Notes

1. *logical-DBname* was passed as the first parameter to the SQL_OPEN statement used to open the database.
2. If several databases have been opened simultaneously, the last database opened is taken as the default.
3. To go from one database to another, we suggest using the NS_FUNCTION CHANGEDBCNTX command because the AT command may no longer be supported in future releases.

Example

```
SQL_OPEN "BASE1" , "SCOTT/TIGER@LOCAL1DB2"
SQL_OPEN "BASE2" , "SCOTT/TIGER@LOCAL2DB2"
SQL_OPEN "BASE3" , "SCOTT/TIGER@LOCAL3DB2"

SQL_EXEC SELECT...          ; SELECT on BASE3

SQL_EXEC AT BASE2 SELECT... ; SELECT on BASE2
SQL_EXEC AT BASE2 FETCH...         ; FETCH on BASE2

SQL_EXEC FETCH...           ; FETCH on BASE3
```

See also SQL_OPEN, SQL_CLOSE, NS_FUNCTION CHANGEDBCNTX, NS_FUNCTION GETDBNAME, SQL_ERROR%, SQL_ERRMSG$

SQL_EXEC instruction

Executes an SQL command.: SELECT, INSERT, UPDATE, CREATE TABLE …

Syntax **SQL_EXEC** [**AT** *database-name*] *SQL-command* [**USING** *cursor-handle*]

Parameters

database-name	CSTRING	I	logical name of database
SQL-command	CSTRING	I	SQL command to execute
cursor-handle	INT(4)	I	cursor value

Notes

1. The SQL command is passed directly without quotes. It can correspond to any Oracle SQL command, whether it's a data definition command (CREATE TABLE, CREATE INDEX, ...) or a data manipulation command (SELECT, INSERT, UPDATE, ...).
2. The AT command can only be used with databases which allow several simultaneous connections. The query is sent to the database specified after the AT command (without quotes and *case-sensitive*). If the AT command isn't specified, the SQL_EXEC executes on the current database.
3. If USING *cursor-handle* is specified, it indicates which cursor previously opened by SQL_OPENCURSOR% must be used to execute the SQL command. If no cursor has been opened, the cursor's value is that of DEFAULT_CURSOR: -1.
4. The SQL command can return values in NCL variables. For this, just pass these variables in parameters.
5. It is possible to pass a segment's field as a data-receiving variable in an SQL query.
6. The commands SQL_EXEC, SQL_EXECSTR and SQL_EXEC_LONGSTR depend on the SQL language accepted by the DBMS in use (Refer to the DBMS documentation).
7. For SQL commands that are too long, it is possible to use the special continuation character "\":
```
SQL_EXEC UPDATE SAMPLE SET COMPAGNY = :A$\
                        WHERE TOWN = :C$\ AND \
                              COUNTRY  = :D$
```
8. The types of variables recognized by the interface are:
   - INT(1), INT(2), and INT(4),
   - NUM(8), NUM(4),
   - STRING,
   - CSTRING,
   - CHAR.

9. Each database has its own implementation of SQL. In your DBMS documentation, refer to the topics concerning your database for more information about the conversion of NCL types to authorized SQL types.

10. The INTO clause is used by the SELECT and FETCH commands. It defines a list of host variables. Its syntax is:

```
INTO :var1 [:indic1] [, :var2 [:indic2] \
[, ... ] ]
```

11. We suggest using INTO in a SELECT to improve performance because during a FETCH, in each loop, the driver has to analyze the variables of the INTO clause. Using the INTO clause in a FETCH should be restricted to doing things like be entering elements into an array.

12. Always put a ":" before the name of a variable or flag.

13. A flag is an NCL integer variable which can have the following values:

- ♦ NULL_VALUE_INDICATOR (i.e. -1) indicates that the associated NCL variable which precedes it has a NULL value.
- ♦ Any other value indicates that the associated NCL variable which precedes it has a NOT NULL value, and can therefore be used.

14. In SQL, NULL does not mean 0 or an empty string (""). However, to make it possible to assign a value in all cases, when a column contains a NULL value, a numeric target NCL variable will be assigned a 0 and a string target NCL variable will be assigned an empty string ("").

Example

```
LOCAL CODE%,I%,AGE%,IND1%,IND2%
LOCAL COUNTRY$,CITY$,A$,B$
LOCAL TCODE%[10]
LOCAL TCOUNTRY$[10]

CITY$ = "NEW YORK"

; ========
;  1st example
; ========
; ---- Select a subset
SQL_EXEC SELECT CODE,COUNTRY \
         FROM WORLD \
         WHERE TOWN =:CITY$
; ---- Read the first to last entry
WHILE SQL_ERROR% = 0
  SQL_EXEC FETCH INTO :CODE%,:COUNTRY$
  IF SQL_ERROR% = 0
     INSERT AT END CODE% && COUNTRY$ TO LBOX1
  ENDIF
ENDWHILE

; ============================
;  2nd example(most efficient)
; ============================
; ---- Select a subset
;      and read the first entry
SQL_EXEC SELECT CODE,COUNTRY \
         FROM WORLD \
         INTO:CODE%,:COUNTRY$ \
         WHERE TOWN =:CITY$
; ---- Read the second to the last entry
WHILE SQL_ERROR% = 0
  INSERT AT END CODE% && COUNTRY$ TO LBOX1
  SQL_EXEC FETCH
ENDWHILE

; =========
;  3rd example
; =========
; ---- Select a subset
SQL_EXEC SELECT CODE,COUNTRY \
         FROM WORLD \
         WHERE TOWN =:CITY$
; ---- Read 1st entry to last entry
;      by filling TCODE% and TCOUNTRY$ tables
I% = 0
WHILE (SQL_ERROR% = 0) AND (I% < 10)
  SQL_EXEC FETCH INTO :TCODE%[I%],:TCOUNTRY$[I%]
  I% = I% + 1
ENDWHILE


; =============================
;  Using flags
; =============================
SQL_EXEC CREATE TABLE FAMILY( NAME       VARCHAR2(10),\
                              AGE        NUMBER,      \
                              CHILDNAME  VARCHAR2(10))
FATHER$ = "STEVE"
AGE%    = 35
SON$    = "PETER"
IND1%   = 0
IND2%   = 0
; --- Insert "STEVE",35,"PETER" into table
SQL_EXEC INSERT INTO FAMILY\
                VALUES (:FATHER$:IND1%, :AGE%, :SON$:IND2%)

FATHER$ = "PETER"
AGE%  = 10
IND1% = 0
IND2% = NULL_VALUE_INDICATOR
; --- Insert "PETER",10,NULL into table
SQL_EXEC INSERT INTO FAMILY VALUES (:FATHER$:IND1%, :AGE%, :SON$:IND2%)

; ---- The SELECT loop places the listbox LBOX
;      'STEVE's son is PETER'
;      'PETER does not have a son.'
SQL_EXEC SELECT NAME, AGE, CHILDNAME\
         INTO:FATHER$:IND1%,:AGE%,:SON$:IND2% \
         FROM FAMILY
WHILE SQL_ERROR% = 0
  ; ---- IND1% is always 0 here
  IF IND2% = -1
     INSERT AT END FATHER$ & "does not have a son." TO LBOX
  ELSE
     INSERT AT END FATHER$ & "'s son "  &\
                   "is" & SON$ TO LBOX
  ENDIF
  SQL_EXEC FETCH
ENDWHILE
```

See also SQL_EXECSTR, SQL_EXEC_LONGSTR, SQL_ERROR%, SQL_ERRMSG$

SQL_EXECSTR instruction

Executes an SQL command: SELECT, INSERT, UPDATE, CREATE TABLE …

Syntax **SQL_EXECSTR** *SQL-command* [, *variable* [, *variable* [, ...]]] [**USING** *handle-name*]

Parameters

SQL-command	CSTRING	I	SQL order to execute
variable		I	NCL variable list
handle_name	INT(4)	I	cursor value

Notes

1. *SQL-command* is either a string *host* variable or a character string containing the SQL command to execute in quotation marks.
2. When you use the SQL_EXEC instruction, you write the names of the *host* variables directly in the text of the SQL query. When you use the SQL_EXECSTR instruction, the *host* variables are parameters of the instruction.
3. When you use the SQL_EXECSTR instruction, each *host* variable is represented in the text of the query by a ":" character. The first ":" corresponds to the first *host* variable passed as a parameter, and so on.
4. The other functions of the SQL_EXECSTR command are the same as SQL_EXEC.

Example

```
LOCAL REQ$, TABLE$, FATHER$, SON$
LOCAL AGE%, IND1%, IND2%, CURS1%

TABLE$   = "FAMILY"
AGE%     = 20
REQ$ = "SELECT NAME, AGE, CHILDNAME INTO : :,:,: : FROM '" &\
            TABLE$ & "' WHERE AGE > :"

; ---- Open a cursor
CURS1%=SQL_OPENCURSOR%

; ---- Select persons older than 20 from
;      the FAMILY table
SQL_EXECSTR :REQ$, :FATHER$,:IND1%,:AGE%,:SON$,:IND2%,:AGE%, USING CURS1%

WHILE SQL_ERROR% = 0
  IF IND2% = NULL_VALUE_INDICATOR
     INSERT AT END FATHER$ & " does not have a son" TO LBOX
  ELSE
     INSERT AT END FATHER$ & "'s son" &  "is" & SON$ TO LBOX
  ENDIF
  SQL_EXEC FETCH USING CURS1%
ENDWHILE

; ---- Close the cursor
```

```
SQL_CLOSECURSOR
```

See also SQL_EXEC, SQL_EXEC_LONGSTR, SQL_OPENCURSOR%, SQL_CLOSECURSOR, SQL_ERROR%, SQL_ERRMSG$

SQL_EXEC_LONGSTR instruction

Executes an SQL command: SELECT, INSERT, UPDATE, CREATE TABLE …

Syntax **SQL_EXEC_LONGSTR** *sql-string-address, var-array-address, cursor-num*

Parameters

sql-string-address	INT(4)	I	address of the character string containing the SQL statement to execute
var-array-address	INT(4)	I	address of the array containing the host variables (or indicators)
cursor-num	INT(2)	I	cursor value

Notes

1. The executed statement can contain any SQL command in the host language (DML or DDL). The size of the string depends on the RDBMS used; it is unlimited for certain database engines and limited to 4096 characters for others.
2. *sql-string-address* is the address of the string which contains the SQL command to execute.
3. *var-array-address* is an array of NCLVAR segments which describe the NCL host variables. If your SQL statement does not use any variables, pass 0 in *var-array-address*.
4. When you use the SQL_EXEC_LONGSTR instruction, each *host* variable is represented in the text of the query by a ":" character. The first ":" corresponds to the first *host* variable in the array of *host* variables, and so on.
5. The NCLVAR segment and any constants used are declared in the NSDBMS library as follows:

```
SEGMENT NCLVAR
  INT     PTR_VAR(4)
  INT     TYPE_VAR(2)
  INTEGER SIZE_VAR
  INT     RESERVED(4)
ENDSEGMENT

CONST TYPE_SQL_INT%                 0
CONST TYPE_SQL_STRING%              1
CONST TYPE_SQL_CSTRING%             2
CONST TYPE_SQL_NUM%                 3
CONST TYPE_SQL_SEGMENT%             10
CONST TYPE_SQL_IMAGE%               11
CONST TYPE_SQL_SELECT_BLOB%         12
CONST TYPE_SQL_INSERT_BLOB%         13
```

6. This array of segments should have an **index that is greater than** the number of variables used (the last element contains 0). You are therefore advised to fill this array initially (using the NCL FILL verb) to ensure that element 0 actually exists, since the end of the scan is determined by this element.
7. If no cursors have been opened, the cursor value must be set to that of the DEFAULT CURSOR: -1.
8. SQL_EXEC_LONGSTR replaces SQL_EXECLONGSTR%. To use this instruction, you will still find the code you need in the notes of NSDBMS.NCL.
9. The other function of SQL_EXEC_LONGSTR instruction are the same as SQL_EXEC.

Example

```
LOCAL NCLVAR VARLIST[3]      ; for 2 variables
LOCAL SQL_STR$       ; string to pass
LOCAL VAR1%, VAR2$   ; host variables
LOCAL CONDITION%     ; input variable

; ---- Set the array to 0
FILL @VARLIST, SIZEOF VARLIST, 0

SQL_STR$            = "SELECT VCHAR, VINT " & "FROM TAB1 " &\ "WHERE VINT >= :"

VARLIST[0].PTR_VAR  = @CONDITION%
varlist[0].TYPE_VAR = TYPE_SQL_INT%
VARLIST[0].SIZE_VAR = SIZEOF @CONDITION%

SQL_EXEC_LONGSTR @SQL_STR$, @VARLIST, DEFAULT_CURSOR
FILL @VARLIST, SIZEOF VARLIST, 0
SQL_STR$ = "FETCH INTO :, :"

VARLIST[0].PTR_VAR  = @var2$
VARLIST[0].TYPE_VAR = TYPE_SQL_CSTRING%
VARLIST[0].SIZE_VAR = SIZEOF var2$
VARLIST[1].PTR_VAR  = @var1%
VARLIST[1].TYPE_VAR = TYPE_SQL_INT%
VARLIST[1].SIZE_VAR = SIZEOF var1%

WHILE SQL_ERROR% = 0

  SQL_EXEC_LONGSTR @SQL_STR$, @VARLIST, DEFAULT_CURSOR
  IF SQL_ERROR% = 0
     MESSAGE "SELECT", VAR1% && VAR2$
  ENDIF
ENDWHILE
```

See also FILL (NCL), NSDBMS.NCL, SQL_EXEC, SQL_EXECSTR, SQL_ERROR%, SQL_ERRMSG$

SQL_OPENCURSOR% function

Opens a cursor and returns its handle.

Syntax	**SQL_OPENCURSOR%**
Returned value	INT(4)

Notes

1. After opening the cursor, it can be used with the following instructions:
   ```
   SQL_EXEC SELECT ... USING cursor-handle
   SQL_EXEC FETCH  ... USING cursor-handle
   ```
2. A cursor is an internal resource managed by the NAT SYSTEM DLL and is used, for example, to store the current table row position for the next SQL call.
3. When the system is opened, only one cursor is defined, known as the DEFAULT_CURSOR.
4. If no cursors have been opened, this DEFAULT_CURSOR will be used to execute all SQL statements that maintain current positions within the database, including SELECT and FETCH statements.
5. A problem occurs if an SQL statement other than FETCH (for example UPDATE or INSERT) is embedded in a scanning sequence; the current position is lost and the FETCH statement that follows the embedded statement will terminate with the error.

   SQL_OPENCURSOR% solves this problem by executing all SELECT and FETCH commands with the new cursor.
6. Generally speaking, a new cursor should be opened each time you wish to perform a SELECT FETCH scan while another similar scan is still in progress with the last cursor opened.
7. The NAT SYSTEM DLL specifically designed for the DBMS stores cursors in a LIFO (Last In First Out) stack: SQL_OPENCURSOR% stacks and SQL_CLOSECURSOR unstacks.
8. The following rules apply when executing a statement with a cursor:
   - Statements are always executed with the specified cursor.
   - If with SQL_EXEC, the USING clause isn't specified, the commands are executed with the DEFAULT_CURSOR.
9. When several databases are opened simultaneously, the cursor opened by SQL_OPENCURSOR% is immediately associated with the current database.
10. If you want to open a cursor in a database other than the current one, you must execute the SQL_EXEC CHANGEDBCNTX:*otherbase$* command to change databases before you execute SQL_OPENCURSOR%.

Example

See the example of the SQL_CLOSETHECURSOR instruction.

See also SQL_CLOSECURSOR, SQL_OPENTHECURSOR%, SQL_CLOSETHECURSOR, SQL_ERROR%, SQL_ERRMSG$

SQL_CLOSECURSOR instruction

Closes the last cursor opened and the last occupied by SQL_OPENCURSOR%.

Syntax **SQL_CLOSECURSOR**

Notes

1. SQL_CLOSECURSOR closes the last cursor opened, situated at the top of the LIFO (Last In First Out) cursor stack.
2. SQL_CLOSECURSOR must only close cursors opened with SQL_OPENCURSOR%.
3. The error codes returned by SQL_ERROR% for this instruction are: -32003 or -32005.
4. The SQL_CLOSECURSOR instruction must not be used with the IM module of NatStar.
5. Nat System recommends you to use SQL_CLOSETHECURSOR instead of SQL_CLOSECURSOR.

Example

See the example of the SQL_CLOSETHECURSOR instruction.

See also SQL_OPENCURSOR%, SQL_OPENTHECURSOR%, SQL_CLOSETHECURSOR, SQL_ERROR%, SQL_ERRMSG$

SQL_OPENTHECURSOR% function

Opens a cursor and returns its handle.

Syntax **SQL_OPENTHECURSOR%**

Returned value INT(2)

Notes

1. After opening the cursor, it can be used with the following instructions:
   ```
   SQL_EXEC SELECT ... USING cursor-handle
   SQL_EXEC FETCH  ... USING cursor-handle
   ```
2. A cursor is an internal resource managed by the NAT SYSTEM DLL and is used, for example, to store the current table row position for the next SQL call.
3. When the system is opened, only one cursor is defined, known as the DEFAULT_CURSOR.
4. If no cursors have been opened, this DEFAULT_CURSOR will be used to execute all SQL statements that maintain current positions within the database, including SELECT and FETCH statements.
5. A problem occurs if an SQL statement other than FETCH (for example UPDATE or INSERT) is embedded in a scanning sequence; the current position is lost and the FETCH statement that follows the embedded statement will terminate with the error.

   SQL_OPENCURSOR% solves this problem by executing all SELECT and FETCH commands with the new cursor.
6. Generally speaking, a new cursor should be opened each time you wish to perform a SELECT FETCH scan while another similar scan is still in progress with the last cursor opened.
7. The following rules apply when executing a statement with a cursor:
   - Statements are always executed with the specified cursor.
   - If with SQL_EXEC, the USING clause isn't specified, the commands are executed with the DEFAULT_CURSOR.
8. When opening several databases at the same time, the cursor opened by SQL_OPENTHECURSOR% is immediately associated with the current database.
9. If you want to open a cursor in a database other than the current one, you must execute the SQL_EXEC CHANGEDBCNTX:*otherbase$* command to change databases before you execute SQL_OPENCURSOR%.

Example

Refer to the example of the SQL_CLOSETHECURSOR instruction.

See also SQL_OPENCURSOR%, SQL_CLOSECURSOR, SQL_CLOSETHECURSOR, SQL_ERROR%, SQL_ERRMSG$

SQL_CLOSETHECURSOR instruction

Closes the cursor associated with the given handle.

Syntax **SQL_CLOSETHECURSOR** *cursor-handle*

Parameter

cursor-handle	INT(4)	I	handle of cursor to close

Note

1. SQL_CLOSETHECURSOR can only close cursors opened with SQL_OPENTHECURSOR%.

Example

```
; ---- Example showing the two different types of
;      cursors (for clarity, we have not
;      included error test code)

SQL_EXEC ....                      ; uses the default cursor
C1% = SQL_OPENCURSOR%              ; opens the C1% cursor
SQL_EXEC UPDATE ...                ; uses the default cursor
SQL_EXEC SELECT ...                ; uses the default cursor
SQL_CLOSETHECURSOR C1%             ; => error
C2% = SQL_OPENTHECURSOR%           ; opens the C2% cursor
SQL_EXEC UPDATE ...                ; uses the default cursor
SQL_EXEC UPDATE ... USING C1%      ; uses the C1% cursor
SQL_EXEC SELECT ... USING C2%      ; uses the C2% cursor
SQL_EXEC SELECT ... USING C1%      ; uses the C1% cursor
SQL_CLOSECURSOR                    ; closes the C1% cursor
SQL_EXEC UPDATE ....               ; uses the default cursor
SQL_EXEC SELECT .... USING C2%     ; uses the C2% cursor
SQL_CLOSECURSOR%                   ; => error
SQL_CLOSETHECURSOR C2%             ; closes the C2% cursor
SQL_EXEC ....                      ; uses the default cursor
```

See also SQL_OPENCURSOR%, SQL_CLOSECURSOR, SQL_OPENTHECURSOR%, SQL_ERROR%, SQL_ERRMSG$

SQL_ERROR% function

Returns the error code of the last SQL_ instruction executed.

Syntax **SQL_ERROR%**

Returned value INT(4)

Notes

1. SQL_ERROR% complies with SQL conventions. The function returns:
   - 0 if no errors occurred.
   - A positive number for non-fatal errors (the instruction was executed but issued a warning).
   - A negative number for fatal errors (the instruction could not be executed).
2. This function applies to all DB2x driver functions.
3. There are two types of errors returned:
   - Proprietary DBMS SQL error codes which are described in the editor's manuals.
   - Internal NAT SYSTEM error codes. They correspond to errors not handles by the host DBMS. These error messages are numbered and have the format "32XXX".

   Example :
   ```
   -32004 "NSSQLE004 ** NO MORE CURSORS AVAILABLE"
   ```
4. List of internal Nat System errors common to DB2x :

   0 "NSSQLE000 ** UNUSED NATSYS ERROR CODE"

   +100 "NSSQLW100 ** NO ROW WAS FOUND OR LAST ROW REACHED"

   -32001 "NSSQLE001 ** HEAP ALLOCATION ERROR"

   Cause : Internal memory allocation/deallocation error

   -32002 "NSSQLE002 ** DYNAMIC ALLOCATION ERROR"

   Cause : Internal memory allocation/deallocation error

   -32003 "NSSQLE003 ** DYNAMIC FREE STORAGE ERROR"

   Cause : Internal memory allocation/deallocation error

   -32004 "NSSQLE004 ** NO MORE CURSORS AVAILABLE"

   Cause : Too many cursors opened simultaneously.

   -32005 "NSSQLE005 ** NO MORE CURSORS OR TRYING TO DEALLOCATE ONLY CURSOR"

-32006 "NSSQLE006 ** INVALID INTO CLAUSE in FETCH/SELECT"

Cause : Syntax error in the INTO clause of a SELECT or a FETCH statement.

-32007 "NSSQLE007 ** TOO MANY VARIABLES IN INTO CLAUSE"

Cause : More than 200 in the INTO clause

-32008 "NSSQLE008 ** MISSING HOST VARIABLE AFTER ','"

Cause : Syntax error in the INTO clause. Variable missing after a continuation comma.

+32009 "NSSQLW009 ** INTO CLAUSE : NOT ENOUGH VARIABLES"

Cause : A SELECT statement contains an INTO clause with fewer variables than the number of variables returned by the query.

Warning : The system will still fill the host variables supplied to it.

+32010 "NSSQLW010 ** AN OPENED CURSOR WAS CLOSED BY SYSTEM"

Cause : Following the arrival of a new SQL command for a cursor, the system forced the closure of a cursor containing an active query.

-32011 "NSSQLE011 ** WHERE/VALUE CLAUSE : NOT ENOUGH VARIABLES"

Cause : Not enough host variables received in the table to be able to substitute the variables specified in the WHERE clause.

-32012 "NSSQLE012 ** INVALID INPUT VARIABLE DATA TYPE"

Cause : Invalid data type in a WHERE clause.

-32013 "NSSQLE013 ** MISSING 'OF' AFTER 'WHERE CURRENT'

Cause : Syntax error in UPDATE WHERE CURRENT OF.

-32014 "NSSQLE014 ** NO OUTPUT VARIABLES DEFINED FOR FETCH"

Cause : The FETCH and the prior SELECT have not defined any output variables (INTO clause).

-32015 "NSSQLE015 ** CURSOR NOT READY (MISSING SELECT) "

Cause : FETCH attempted without a prior SELECT or cursor closed by the system between the SELECT and the FETCH statements.

-32016 "NSSQLE016 ** INVALID SQL DATA TYPE"

Cause : Data type invalid for output.

-32017 "NSSQLE017 ** INVALID DATA CONVERSION REQUESTED"

Cause : Type conversion invalid for output.

STRING -> NUM

NUM -> STRING

REAL -> INTEGER

INTEGER -> REAL

-32018 "NSSQLE018 ** NUMERIC DATA TYPE : INVALID LENGTH"

Cause : Invalid length for the data type (for example, real number with a length of 48)

-**32019** "NSSQLE019 ** INVALID DECIMAL PACKED FORMAT"

Cause : Unable to convert data to packed decimal format.

+32020 "NSSQLW020 ** STRING DATA TRUNCATED"

Cause : The string passed as a variable is shorter than the field received from the DBMS. The received field has been truncated.

-32021 "NSSQLE021 ** RESET STORAGE ERROR"

Cause : Deallocation error of the internl heap

+32022 "NSSQLW022 ** FUNCTION NOT SUPPORTED IN DB2 DATABASE"

Cause : The executed instruction is not available.

-32023 "NSSQLE023 ** TOO MANY OPENED DATABASES"

Cause : More than 5 databases opened simultaneously.

+32024 "NSSQLW024 ** DB ALREADY OPENED"

Cause : The database used with SQL_OPEN has already been opened.

-32025 "NSSQLE025 ** DB NOT PREVIOUSLY OPENED"

Cause : Attempt to close a database that has not been happened.

-32026 "NSSQLE026 ** INVALID DATABASE NAME REF"

Cause : Unknown database name used in the AT clause of the SQL_EXEC statement (database not opened)

-32028 "NSSQLE028 ** UNABLE TO GET DB2 LOGIN"

Cause : Failed to connect to DB2 (e.g. server name error).

-32029 "NSSQLE029 ** DB2 VARIABLE INPUT BIND FAILED"

Cause : Type mismatch between a variable and a database field.

-32030 "NSSQLE030 ** DB2 VARIABLE OUTPUT BIND FAILED"

Cause : Type mismatch between a variable and a database field.

-32031 "NSSQLE031 ** DB2 BUFFER FILL ERROR"

Cause : Buffer overflow (due to data conversion,etc.)

-32032 "NSSQLE032 ** RPC PARAMETER NAME EXPECTED"

Cause : Remote procedure call : procedure name missing

-32033 "NSSQLE033 ** TOO MANY RPC PARAMETERS"

Cause : Remote procedure call : too many parameters specified

-32034 "NSSQLE034 ** RPC PROCEDURE NAME EXPECTED"

Cause : Remote procedure call : procedure name missing

-32035 "NSSQLE035 ** NOT ENOUGH PARAMETERS FOR RPC CALLS"

Cause : Remote procedure call : parameters missing

-32036 "NSSQLE036 ** INVALID RPC PARAMETERS SUPPLIED"

Cause : Remote procedure call : invalid parameters specified

-32037 "NSSQLE037 ** INVALID RPC PROCEDURE INITIALIZATION"

-32038 "NSSQLE038 ** RPC PROCEDURE EXECUTION FAILED"

-32039 "NSSQLE039 ** MEMORY CONSISTENT ERROR"

-32040 "NSSQLE040 ** INVALID TYPE FOR INDICATOR"

-32041 "NSSQLE041 ** CONTEXT NOT CREATED"

-32042 "NSSQLE042 ** CONTEXT NOT FOUND"

-32044 "NSSQLE044 ** NO SET LOGIN TIME"

-32045 "NSSQLE045 ** NO SET TIME"

-32046 "NSSQLE046 ** SET MAXPROCS FAILED"

-32047 "NSSQLE047 ** DB OPEN FAILED"

-32048 "NSSQLE048 ** DB NOT OPENED"

-32049 "NSSQLE049 ** LOGIN RECORD NOT ALLOCATED"

-32050 "NSSQLE050 ** MEMORY DEALLOCATION ERROR"

-32051 "NSSQLE051 ** CURSOR NOT FOUND"

-32052 "NSSQLE052 ** MUST EXECUTE SELECT BEFORE THE FETCH COMMAND"

-32053 "NSSQLE053 ** ERROR IN CLOSING DATABASE"

-32054 "NSSQLE054 ** ERROR IN EXECUTING SQL STATEMENT"

-32055 "NSSQLE055 ** ERROR IN EXECUTING FETCH COMMAND"

-32056 "NSSQLE056 ** INDICATOR'S SIZE TOO SMALL TO HOLD VALUE"

-32057 "NSSQLE057 ** UNKNOWN NCL VARIABLE TYPE PASSED"

-32058 "NSSQLE058 ** RPC : INIT ERROR"

-32059 "NSSQLE059 ** RPC : PARAMETERS FOUND BUT NO VALUES CLAUSE"

-32060 "NSSQLE060 ** RPC : PARAMETER TYPE MISMATCH"

-32061 "NSSQLE061 ** RPC : PROCEDURE NAME MISSING"

-32062 "NSSQLE062 ** RPC : INDICATORS MAY ONLY BE ON OUT VARIABLES"

-32063 "NSSQLE063 ** RPC : SQL SERVER ERROR DURING RPC PREPARATION"

-32064 "NSSQLE064 ** RPC : SQL SERVER ERROR DURING RPC EXECUTION"

-32065 "NSSQLE065 ** RPC : SQL SERVER ERROR DURING RPC EXEC CHECK"

-32066 "NSSQLE066 ** RPC : PROCEDURE NOT PREPARED"

-32067 "NSSQLE067 ** LOGGER : CAN'T OPEN FILE"

-32068 "NSSQLE068 ** PARSER : TOKEN TABLE FULL"

-32069 "NSSQLE069 ** EXEC : INCOMPATIBLE CURSOR MODE"

-32070 "NSSQLE070 ** EXEC : SQL SERVER ERROR DURING SIZE BUFFERING EXECUTION"

-32071 "NSSQLE071 ** EXEC : SQL SERVER ERROR DURING SIZE BUFFERING DELETION"

-32072 "NSSQLE072 ** EXEC : INVALID CURSOR MODE"

-32073 "NSSQLE073 ** EXEC : THAT ROW IS NOT IN BUFFER"

-32074 "NSSQLE074 ** EXEC : INCORRECT SYNTAX FOR THIS CURSOR MODE"

-32075 "NSSQLE075 ** EXEC : MISSING INTO CLAUSE FOR THIS CURSOR MODE"

-32076 "NSSQLE076 ** EXEC : INVALID SIZE FOR ROW BUFFERING"

-32077 "NSSQLE077 ** EXEC : INVALID ROW NUMBER"

-32078 "NSSQLE078 ** EXEC : MEMORY DEALLOCATION ERROR FOR SCROLL STATUS"

-32079 "NSSQLE079 ** EXEC : SQL SERVER : ROW IS MISSING IN SCROLL BUFFER"

-32080 "NSSQLE080 ** NO STATEMENT IN PROGRESS"

-32081 "NSSQLE081 ** DATA NOT READY TO RESULT PROCESSING"

-32082 "NSSQLE082 ** INVALID WINDOW HANDLE"

-32083 "NSSQLE083 ** USER MESSAGE MUST BE RANGE IN 0 AND 15"

-32084 "NSSQLE084 ** INVALID STATEMENT SEND TO DLL"

-32085 "NSSQLE085 ** NO MORE RESULT TO FETCH"

-32086 "NSSQLE086 ** INVALID PARAMETER TO CHANGE OPTION"

-32087 "NSSQLE087 ** INVALID PARAMETER TO CHANGE OPTION VALUE"

-32088 "NSSQLE088 ** LOGIN TIME CHANGE FAILED"

-32089 "NSSQLE089 ** TIMEOUT CHANGE FAILED"

-32090 "NSSQLE090 ** INVALID NS_FUNCTION STATEMENT"

-32091 "NSSQLE091 ** INVALID DATABASE NAME"

-32092 "NSSQLE092 ** INVALID INTO CLAUSE WHEN

ASYNCHRONOUS MODE"

-32093 "NSSQLE093 ** INVALID LENGTH FOR DATABASE NAME"

-32095 "NSSQLE095 ** INVALID LENGTH FOR USER NAME"

-32096 "NSSQLE096 ** INVALID LENGTH FOR PASSWORD"

-32097 "NSSQLE097 ** INVALID LENGTH FOR SERVER NAME"

-32098 "NSSQLE098 ** INVALID LENGTH FOR SERVER NAME"

-32099 "NSSQLE099 ** KEYWORD AT IS NOT SUPPORTED"

-32101 "NSSQLE101 ** UNABLE TO OPEN FILE"

-32102 "NSSQLE102 ** NO MEMORY AVAILABLE"

-32103 "NSSQLE103 ** NO CONNECTION AVAILABLE TO UPDATE IMAGE/TEXT"

-32104 "NSSQLE104 ** CONNECTION CLOSED BY SERVER"

-32200 "NSSQLE200 ** COMPUTE RESULT IN PROGRESS"

Example

```
...
SQL_EXEC SELECT ENO,ENAME INTO :NO%,:NAME$ FROM EMPLOYEE
WHILE SQL_ERROR% = 0
  INSERT AT END "NO=" & NO% & " NAME=" & NAME$ TO LBOX1
  SQL_EXEC FETCH INTO :NO%,:NAME$
ENDWHILE
IF SQL_ERROR% < 0
```

```
   MESSAGE "fatal error", SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
ELSE
   IF SQL_ERROR% > 0
      MESSAGE "Warning", SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
   ELSE
       MESSAGE "OK, No error", SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
   ENDIF
ENDIF
...
```

See also ***NSnn_SQL Library Error*** **messages**

SQL_ERRMSG$, NS_FUNCTION ERRORCOUNT, NS_FUNCTION GETERROR, NS_FUNCTION CALLBACK

SQL_ERRMSG$ function

Returns the error message (character string) for the last SQL_ instruction executed.

Syntax	**SQL_ERRMSG$** *(error-code)*			
Parameter	*error-code*	INT(4)	I	error code
Returned value	CSTRING			

Notes

1. SQL_ERRMSG$ returns the last message stored in a work area in the DLL when the error occurred.
2. This function applies to all DB2x driver functions.
3. See SQL_ERROR% for a detailed list of error codes and messages.

Example

See the example of the SQL_ERROR% instruction.

See also

NSnn_SQL Library Error **messages**

SQL_ERROR%, NS_FUNCTION ERRORCOUNT, NS_FUNCTION GETERROR, NS_FUNCTION CALLBACK

SQL_PROC command

Executes a stored procedure.

Syntax **SQL_PROC** *procedure-name*

[SCHEMA].[(@Parameter-name1 [OUT] [,@Parameter-name2 [OUT]...])

VALUES (:Host-Variable1 [,:Host-Variable2...])]

By default, SCHEMA stands for the user name of the connection who has created a stored procedure/function.

Parameter	*procedure-name*	CSTRING I	name of the procedure to call

Notes

1. The SQL_PROC keyword informs the parser that a procedure is being called. The word that follows SQL_PROC specifies the name of the called procedure.
2. If the called procedure has parameters, their names must be specified exactly as they are defined in the procedure.

Parameters	**Syntax**
Input parameter	*parameter-name* in the parameter list.
	parameter-name can be followed by IN which is the default in any case.
Output or Input/Output	*parameter-name* must be followed by OUT
Values of the procedure's Input parameters	after the VALUES keyword, supply a list of host variables containing the values of the parameters.

3. ☞ host variables are preceded by a colon (':').
4. ⚠ **Restrictions** :

- The maximum size allowed for ALPHA host variables (CHAR, VARCHAR, etc) is 255 characters.

- The size of the host variable must match the size of the parameter passed to the procedure. Therefore, when you declare a string host variable, you must also specify its size: `LOCAL A$(15)`.

- If the size of the character string retrieved in the host variable is less than the reserved size, any space that is not occupied by the data item will be padded with blanks (ASCII code 32). The NCL operator, SKIP, can be used to remove any unwanted spaces (cf. *NCL Programming Manual).*

- When a procedure is called with output parameters (OUT), you must use SQL_RETURN to retrieve any errors.

  In this case, you have to use NS_FUNCTION RPCRETCODEON to activate the use of the return code.

  There are FUNCTION or INSTRUCTION type stored procedures.

  You can only use NS_FUNCTION RPCRETCODEON for FUNCTION type stored procedure.

5. The syntax checker does not check anything typed after 'SQL_EXEC'. So, if SQL_PROC is misspelled, a syntax error will not be generated but SQL errors will occur.

Example 1

```
; Execute the RPCTEST procedure below created for
; database on an DB2/AS400 server
; this procedure specifies the OUT variables
;and returns a stream of data (type "SELECT * FROM TABLE")
;supposing that there is no return code
;
; Prototype of the procedure
; CREATE PROCEDURE RPCTEST(PARAM1 INT OUT,
;                          PARAM2 INT OUT,
;                          PARAM3 INT OUT,
;                          PARAM4 INT)

; Call RPCTEST procedure
LOCAL A%, B%, C%, D%, R%
LOCAL A$, B$

A% = 2
B% = 3
C% = 4
D% = 1

SQL_EXEC SQL_PROC RPCTEST (PARAM1 OUT, PARAM2 OUT, \
                           PARAM3 OUT, PARAM4)            \
         VALUES (:A%,:B%,:C%, :D%)
if SQL_ERROR% <> 0
      ; Error handling
ENDIF

; Retrieve the result of the procedure in NCL variables

; Here we retrieve the "Message retrieved by a FETCH" in A$
SQL_EXEC FETCH INTO :A$
IF SQL_ERROR% <> 0
      ; Error handling
ENDIF
```

```
; Retrieve the result of a SELECT (a single column)
WHILE SQL_ERROR% = 0
  SQL_EXEC FETCH INTO :B$
  IF SQL_ERROR% <> 0
      ; Error handling
  ENDIF
ENDWHILE

; No return code therefore no SQL_RETURN
; If you do have a return code, you must
; call RPCRETCODEON before calling the procedure
; Display the OUT parameters of the procedure
; A%, B%, C% have been incremented by 1 (by d%)
MESSAGE "Result",A%&&B%&&C%
```

Example 2

```
LOCAL A%, B%, C%, D%, R%, h%
A% = EF_A
; to get the return code
SQL_EXEC NS_FUNCTION RPCRETCODEON
if SQL_ERROR% <> 0
   MESSAGE "Erreur RPCRETCODEON",SQL_ERROR%&&SQL_ERRMSG$(SQL_ERROR%)
ENDIF
;it works like that
SQL_EXEC SQL_PROC PROCEDURE1 (@A% OUT)VALUES ( :A%  )
if SQL_ERROR% <> 0
   MESSAGE "Erreur SQL_PROC PROCEDURE1",SQL_ERRMSG$(SQL_ERROR%)
ENDIF

; Retrieve the result of the procedure in NCL variables
WHILE SQL_ERROR% = 0
  SQL_EXEC FETCH INTO :B%
  IF SQL_ERROR% <> 0
  IF SQL_ERROR% < 0
         MESSAGE "Erreur",SQL_ERRMSG$(SQL_ERROR%)
      ENDIF
  ELSE
         INSERT AT END 'B% ='&&B% TO LISTBOX1
  ENDIF
ENDWHILE

 ; return code of the procedure
SQL_EXEC SQL_RETURN:R%
if SQL_ERROR% <> 0
   MESSAGE "Erreur SQL_RETURN:R%",SQL_ERRMSG$(SQL_ERROR%)
ENDIF

; Display R% (= 200)
INSERT AT END  "Return code from PROCEDURE1 "&&R% TO LISTBOX1

SQL_EXEC NS_FUNCTION RPCRETCODEOFF
if SQL_ERROR% <> 0
   MESSAGE "Erreur RPCRETCODEON",SQL_ERROR%&&SQL_ERRMSG$(SQL_ERROR%)
ENDIF
```

See also SQL_RETURN, NS_FUNCTION RPCRETCODEON, NS_FUNCTION RPCRETCODEOFF, SQL_ERROR%, SQL_ERRMSG$

SQL_RETURN command

Retrieves the error code from the last procedure executed by the SQL_EXEC SQL_PROC command.

Syntax **SQL_RETURN** : *return-code*

Parameter

return-code	INT(4)	O	return code

Note

1. If the procedure was executed unsuccessfully, a negative error code (between -1 and -n) will be returned. This error code depends on the client libraries used.
2. This command MUST be used when you call procedures with OUT parameters (cf. SQL_PROC command).

   In this case, you have to use NS_FUNCTION RPCRETCODEON to activate the use of the return code.

   There are FUNCTION or INSTRUCTION type stored procedures.

   You can only use NS_FUNCTION RPCRETCODEON for FUNCTION type stored procedure.

Example

See the example of the SQL_PROC instruction.

See also SQL_PROC, NS_FUNCTION RPCRETCODEON, NS_FUNCTION RPCRETCODEOFF, SQL_ERROR%, SQL_ERRMSG$

RECORD, REEXECUTE commands

The RECORD command records an SQL sequence so that it can be re-executed using the REEXECUTE command. When you call REEXECUTE, you only supply new values.

Syntax **RECORD** *SQL-statement*

and

REEXECUTE

Parameter	*SQL-statement*	CSTRING	I	SQL sequence to record

Notes

1. The parameters in the SQL sequence must still be accessible when the command 'SQL_EXEC REEXECUTE' is issued.
2. After RECORD, any other SQL statement other than REEXECUTE cancels the current RECORD operation.
3. Using RECORD and REEXECUTE allows you to combine the adaptability of dynamic SQL with the speed of static SQL. In fact, a dynamic SQL order is used when the RECORD command is executed. When the REEXECUTE command is executed, as the analysis of query has already been done by the motor, only the values of the host variables are set.

Example

```
LOCAL CODE%
LOCAL NAME$(25)

SQL_EXEC CREATE TABLE EMP(EMPNO INTEGER,ENAME CHAR(25))

CODE = 1
NOM$ = "NAME1"
SQL_EXEC RECORD INSERT INTO EMP VALUES (:CODE%,:NAME$)
FOR I= 2 TO 100
    CODE = I
    NAME$ = "NAME" & I
    SQL_EXEC REEXECUTE
ENDFOR

; ---- The EMP table now contains
; (  1, "NAME1")
; (  2, "NAME2")
; (  3, "NAME3")
; ...
; ( 99, "NAME99")
; (100, "NAME100")
```

TYPE_SQL_INSERT_BLOB%, TYPE_SQL_SELECT_BLOB% types for blobs

Enables management of binary large objects, larger than 32K but whose size remains limited by the DBMS.

Notes

1. Two new NCL data types have been added to NSDBMS.NCL and are to be declared in the Type_Var field of the NCLVAR structure:
   - **TYPE_SQL_INSERT_BLOB%**
   - **TYPE_SQL_SELECT_BLOB%**

   They are used for :
   - inserting a binary file into the database,
   - retrieving a binary file from the database
2. Images inserted with TYPE_SQL_INSERT_BLOB% are not limited in size.
3. Selection of images with TYPE_SQL_SELECT_BLOB% is limited by default to 1 Mb. However, this size may be changed using the NS_FUNCTION CHANGEOPTION TEXTSIZE instruction.

Example

```
LOCAL NCLVAR HL[2]
LOCAL INT IMAGNO
LOCAL DESCRIP$
LOCAL FIMAGE$
LOCAL INT J
LOCAL SQL$
LOCAL BMP%
LOCAL CURSORMODE%
LOCAL Opt$
LOCAL Val%, i%
LOCAL CSTRING Req$(2000)

;Insertion of images with DB_DB21_CURSOR_BINDING
CURSORMODE% = DB_DB21_CURSOR_BINDING ;3
SQL_EXEC NS_FUNCTION SETCURSORMODE :CURSORMODE%
SQL_EXEC DROP TABLE BIGIMAGE
IF SQL_ERROR% <> 0
    MESSAGE "Error ",SQL_ERRMSG$(SQL_ERROR%)
 ENDIF
SQL_EXEC NS_FUNCTION Statement INTO :Req$
IF SQL_ERROR% <> 0
    MESSAGE "Error ",SQL_ERRMSG$(SQL_ERROR%)
 ENDIF
Insert AT END Req$ TO LISTBOX1
; Creation of the table
SQL_EXEC CREATE TABLE BIGIMAGE(NUMERO INTEGER, DESCRIPTION VARCHAR(255),
COLIMAGE BLOB(1000000))
IF SQL_ERROR% <> 0
    MESSAGE "Error ",SQL_ERRMSG$(SQL_ERROR%)
 ENDIF
SQL_EXEC NS_FUNCTION Statement INTO :Req$
Insert AT END Req$ TO LISTBOX1


FILL @HL, SIZEOF HL, 0
FIMAGE$ = F_IMAGE
HL[0].PTR_VAR = @FIMAGE$
HL[0].TYPE_VAR = TYPE_SQL_INSERT_BLOB%
HL[0].SIZE_VAR = SIZEOF FIMAGE$


SQL$="INSERT INTO BIGIMAGE (NUMERO, DESCRIPTION, COLIMAGE) VALUES ( 1,'This is \
a big picture > 32000 bytes', : )"
SQL_EXEC_LONGSTR @SQL$, @HL, DEFAULT_CURSOR
IF SQL_ERROR% <> 0
    MESSAGE "Error ",SQL$&&SQL_ERRMSG$(SQL_ERROR%)
ENDIF
SQL_EXEC NS_FUNCTION Statement INTO :Req$
Insert AT END Req$ TO LISTBOX1


; ---- SELECT with automatic writing  in file EXTRACT.BMP
; ---- One & One column only in the SELECT clause , the IMAGE !!!!!
FILL @HL, SIZEOF HL, 0


FIMAGE$ = "C:\TEMP\EXTRACT.BMP"
FERASE FIMAGE$
HL[0].PTR_VAR = @FIMAGE$
HL[0].TYPE_VAR = TYPE_SQL_SELECT_BLOB%
HL[0].SIZE_VAR = SIZEOF FIMAGE$


SQL$="SELECT COLIMAGE INTO : FROM BIGIMAGE WHERE NUMERO = 1"
SQL_EXEC_LONGSTR @SQL$, @HL, DEFAULT_CURSOR


IF SQL_ERROR% <> 0
    MESSAGE "Error ",SQL$&&SQL_ERRMSG$(SQL_ERROR%)
 ENDIF
SQL_EXEC NS_FUNCTION Statement INTO :Req$
Insert AT END Req$ TO LISTBOX1


; ---- Display of the picture
BMP% = CREATEBMP%(FIMAGE$)
MOVE BMP% TO BMPF


CURSORMODE% = DB_DB21_CURSOR_DEFAULT  ;0
SQL_EXEC NS_FUNCTION SETCURSORMODE :CURSORMODE%
```

See also NSDBMS.NCL, SQL_ERROR%, SQL_ERRMSG$, NS_FUNCTION IMAGEON, NS_FUNCTION IMAGEOFF

UPDATE, CURRENT clauses

Principle

1. **FOR UPDATE OF** is used to stop other users from modifying one or more columns for the duration of a transaction when they have been selected by a given user.

   For example, in SQL pseudo code:

```
; ---- Select and lock the Col-list columns
SELECT ... WHERE ... FOR UPDATE OF Col-list
 ...
 ...  SQL commands modifying contents of Col-list columns
 ...
; ---- End of transaction and unlock columns
COMMIT or ROLLBACK
```

2. **WHERE CURRENT OF** is used to update a record which has been preceded by a SELECT command. This way, for this update command, no calculation of the cursor is required (evaluation of a Where-Clause) for the table, as the cursor is already on the record to update.

   For example, in SQL pseudo code shows the use of the FOR UPDATE OF and WHERE CURRENT OF combination:

```
; ---- Select and lock
BEGIN TRANSACTION (NS_FUNCTION AUTOCOMMITOFF must be set)
SELECT COL1 FROM TABLE1 WHERE COL2 > 10 FOR UPDATE OF COL3
WHILE NOTFOUND% = FALSE
  ; ---- Already on the record to modify
  UPDATE TABLE1 SET COL3 = COL3 + 2 WHERE CURRENT OF
  ; ---- go to next record
  FETCH ...
ENDWHILE

; ---- End of transaction and unlock columns
COMMIT or ROLLBACK
```

Use

1. When using the NSnnDB21 library, the commands are executed dynamically and use are based on the CLI layer. Some SQL commands are not accessible by in *Transact SQL*, which is the case of the CURRENT OF clause
2. When using the NSnnDB2x library, be sure to respect the following rules

   - You must be in a transaction before you can use **FOR UPDATE OF** or **WHERE CURRENT OF**.

   - The command selecting the record must be in the correct format, for example: SELECT ... WHERE ... **FOR UPDATE OF** *Col-list* in order to prevent other users from modifying the current record.

- When the modification is finished, the transaction must finish with a COMMIT or ROLLBACK, to remove the locks on the columns used.
- **FOR UPDATE** applies only to SELECT.
- **CURRENT OF** applies only to UPDATE or DELETE.

Example

```
LOCAL SEN%, BONUS%

;start transactional mode
SQL_EXEC NS_FUNCTION AUTOCOMMITOFF
; ---- Select and lock the columns
SQL_EXEC SELECT SENIORITY, BONUS WHERE SENIORITY >= 2 AND SENIORITY < 25 \
         FOR UPDATE OF BONUS
IF SQL_ERROR% <> 0
; ---- read next record
  SQL_EXEC FETCH INTO :SEN% , : BONUS%
ENDIF
WHILE SQL_ERROR% = 0
  EVALUATE SEN%
    WHERE 2 TO 4 , 6 TO 9
      ; ---- update for 2 to 4 and 6 to 9 years
      BONUS% = BONUS% * 1.02
      SQL_EXEC UPDATE PERSONNEL SET BONUS = :BONUS% WHERE CURRENT OF
ENDWHERE
    WHERE 5,10,15,20 TO
      ; ---- update for 5, 10,15 and 20 years
      BONUS% = BONUS% * 1.03
      SQL_EXEC UPDATE PERSONNEL SET BONUS = :BONUS% WHERE CURRENT OF
ENDWHERE
    ELSE
      ; ---- update other years
      BONUS% = BONUS% * 1.01
      SQL_EXEC UPDATE PERSONNEL SET BONUS = :BONUS% WHERE CURRENT OF
ENDEVALUATE
  ; ---- read next record
  SQL_EXEC FETCH INTO :SEN% , : BONUS%
ENDWHILE

; ---- unlock and commit changes
SQL_EXEC COMMIT
...
```

See also DB2x manuals documentation for more information about locking and transactions.

NS_FUNCTION extensions

Syntax SQL_EXEC **NS_FUNCTION** *command*

Notes

1. *command* is one of the following command.

See also SQL_ERROR%, SQL_ERRMSG$

NS_FUNCTION ANSIOFF, ANSION

☞ NS_FUNCTION ANSIOFF and ANSION have been created to bypass the following problem. The SQL_EXEC UPDATE … WHERE … command fix SQL_ERROR% to 0, even if there is no record.

In the ANSIOFF mode, if an UPDATE or DELETE statement does not affect any records, no errors are returned.

In the ANSION mode, if an UPDATE or DELETE statement does not affect any records, an error (warning) is returned with the code "100".

Syntax **NS_FUNCTION ANSIOFF**

and

NS_FUNCTION ANSION

Notes

1. ANSIOFF is the default mode.
2. SQL_ERROR% enables you to retrieve the warning returned.

Example

```
; ---- ANSIOFF mode by default
SQL_EXEC DELETE ... WHERE ...
; ---- even if no record has been removed, SQL_ERROR% equals zero.

; ---- ANSION mode
SQL_EXEC NS_FUNCTION ANSION
SQL_EXEC UPDATE ... WHERE ...
IF SQL_ERROR% = 100
   MESSAGE "No record updated", \
           SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
ENDIF

; ---- Return to default mode
SQL_EXEC NS_FUNCTION ANSIOFF
```

See also SQL_ERROR%, SQL_ERRMSG$

NS_FUNCTION AUTOCOMMITOFF, AUTOCOMMITON

In AUTOCOMMITON mode, an implicit COMMIT command is automatically executed after each SQL command.

AUTOCOMMITOFF inactivates this mode.

Syntax **NS_FUNCTION AUTOCOMMITOFF**

and

NS_FUNCTION AUTOCOMMITON

Notes

1. AUTOCOMMITON is the default mode.
2. SQL_ERROR% can be used to verify the status of the AUTOCOMMITON or AUTOCOMMITOFF call.
3. After each SQL statement, or group of statements, a COMMIT or ROLLBACK must be executed to confirm or cancel the sequence.
4. The notion of "transactions" becomes operational once AUTOCOMMITOFF has been executed. After each COMMIT or ROLLBACK, a new transaction is started.

Example

```
; start transactional mode
SQL_EXEC NS_FUNCTION AUTOCOMMITOFF
IF SQL_ERROR% <> 0
   MESSAGE "NS_FUNCTION AUTOCOMMITOFF Error", \
           SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
ENDIF

SQL_EXEC INSERT ....
SQL_EXEC INSERT ....
SQL_EXEC INSERT ....
SQL_EXEC INSERT ....

; All INSERTs will be canceled
SQL_EXEC ROLLBACK

; end transactional mode
SQL_EXEC NS_FUNCTION AUTOCOMMITON
IF SQL_ERROR% <> 0
   MESSAGE "NS_FUNCTION AUTOCOMMITOFF Error", \
           SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
ENDIF
SQL_EXEC INSERT ....
; The INSERT is not canceled because the ROLLBACK
; does not have any effect outside of a transaction
SQL_EXEC ROLLBACK
```

See also SQL_ERROR%, SQL_ERRMSG$

NS_FUNCTION CALLBACK

This function allows to reroute the error's messages of DB2 towards a window. Every time an error message appears, DB2 send an event to a window.

Lets you set up centralized management of errors for your application.You no longer need to call SQL_ERROR% and SQL_ERRMSG$ after every command.

Syntax **NS_FUNCTION CALLBACK** *:window-handle,:user-event*

Parameters

window-handle	INT(4)	I	window handle
user-event	INT(4)	I	user event (USER0 - USER15)

Notes

1. ⚠ In UNIX, *window-handle* must use the NAT SYSTEM handle of the window that will receive a notification each time an error occurs.

   For all other targets, *window-handle* must be assigned using the NCL GETCLIENTHWND%(...) function which receives as input the NAT SYSTEM handle of the window that will receive a notification each time an error occurs.
2. To determine the processing carried out, you must program the user event. To obtain the notification of the event in *user-event* must contain 0 for USER0, 1 for USER1,.... or 15 for USER15.
3. To cancel this function, set the window handle to zero.
4. Errors and warnings from the DBMS database being used are returned in their native, proprietary format (see the NSDBMS.NCL file for more information about these formats.)
```
SEGMENT DB_DB2_CLIENT_STRUCT
  INT   errtype(4)
  INT   severity(4)
  CHAR  sqlstate(6)
  INT   msgCode(4)
  INT   nativeCode(4)
  CHAR  msgString(513)
  INT   msgStringLen(2)
ENDSEGMENT
```
5. The type of the error's message is sent to PARAM12%. The handle of the structure is sent to PARAM34%.
6. **If the NS_FUNCTION CALLBACK call is carried out without USING**, the window manages the error's messages by cursor. In this case, the errors of the cursor by default which will be redirected. It is useful to have a window by curosr used.

7. ☞ The error 100 « NO ROW WAS FOUND » is not traced by NS_FUNCTION CALLBACK.

Example

```
LOCAL HDLE_CATCHERR%
LOCAL USER_EVENT%
LOCAL WINDOW_HANDLE%

OPENS CATCHERR,Self%,HDLE_CATCHERR%
MOVE GETCLIENTHWND%(HDLE_CATCHERR%) TO WINDOW_HANDLE%
MOVE 1 TO USER_EVENT%
SQL_EXEC NS_FUNCTION CALLBACK :WINDOW_HANDLE% , :USER_EVENT%
if sql_error% <> 0
  message 'error BODY' , sql_errmsg$(sql_error%)
endif


LOCAL USER_EVENT%
LOCAL WINDOW_HANDLE%
;Stop the callback
MOVE 0 TO WINDOW_HANDLE%
MOVE 1 TO USER_EVENT%
SQL_EXEC NS_FUNCTION CALLBACK :WINDOW_HANDLE% , :USER_EVENT%
; ---------------------------------------------
; In the USER1 event of the CATCHERR window
; ---------------------------------------------
LOCAL MESSAGETYPE%(4)
LOCAL PTR%(4)
MOVE PARAM12% TO MESSAGETYPE%
MOVE PARAM34% TO PTR%

 IF MESSAGETYPE% = CLIENTMSG
  EVALUATE DB_DB2_CLIENT_STRUCT(PTR%).severity
  WHERE DB_DB2_ERROR_WARNING
      INSERT AT END "WARNING : +" &&DB_DB2_CLIENT_STRUCT(PTR%).msgCode TO SELF%
      INSERT AT END "Error Type : +" &&DB_DB2_CLIENT_STRUCT(PTR%).errtype TO \
      SELF%
   INSERT AT END "Native Code " && DB_DB2_CLIENT_STRUCT(PTR%).nativeCode TO \
   SELF%
   IF (DB_DB2_CLIENT_STRUCT(PTR%).MSGSTRINGLEN > 0)
    INSERT AT END "MSGSTRING " & DB_DB2_CLIENT_STRUCT(PTR%).MSGSTRING TO self%
   ENDIF
  ENDWHERE
        WHERE DB_DB2_ERROR_ERROR
   INSERT AT END "ERROR : -" &&DB_DB2_CLIENT_STRUCT(PTR%).msgCode TO SELF%
      INSERT AT END "Error Type : +" &DB_DB2_CLIENT_STRUCT(PTR%).errtype TO \
      SELF%
   INSERT AT END "Native Code " & DB_DB2_CLIENT_STRUCT(PTR%).nativeCode TO self%
   IF (DB_DB2_CLIENT_STRUCT(PTR%).MSGSTRINGLEN > 0)
    INSERT AT END "MSGSTRING " & DB_DB2_CLIENT_STRUCT(PTR%).MSGSTRING TO self%
   ENDIF
  ENDWHERE
  ENDEVALUATE
 ELSE
  INSERT AT END "MESSAGE TYPE UNKNOW" TO SELF%
 ENDIF
```

See also NSDBMS.NCL, SQL_ERROR%, SQL_ERRMSG$, NS_FUNCTION ERRORCOUNT, NS_FUNCTION GETERROR

NS_FUNCTION CHANGEDBCNTX

Sets the current database.

☞ This function has been developed to manage several databases simultaneously.

Syntax	**NS_FUNCTION CHANGEDBCNTX** : *logical-DBname*			
Parameters	*logical-DBname*	CSTRING	I	logical name of the current database

Notes

1. The database specified in *logical-DBname* will become the current database.
2. If the specified database is invalid, the current database will not change.
3. If the SQL_OPENCURSOR command is called after NS_FUNCTION CHANGEDBCNTX, the new cursor will be associated with the database passed as an argument to this function.

Example

```
LOCAL DATABASENAME$

MOVE "PUBS" TO DATABASENAME$

SQL_OPEN "PUBS" , "SCOTT/TIGER@SERVER1"

; current base master
SQL_OPEN "MASTER" , "SCOTT/TIGER@SERVER2"

; current base becomes pubs
SQL_EXEC NS_FUNCTION CHANGEDBCNTX :DATABASENAME$
```

See also SQL_OPEN, SQL_CLOSE, AT

NS_FUNCTION CHANGEOPTION

Modifies the timeout for a SELECT statement or login.

Syntax **NS_FUNCTION CHANGEOPTION** *:parameter, :option*

Parameters

parameter	CSTRING	I	parameter to modify
option	INT(4)	I	value to assign to the parameter

Notes

1. The *option* variable contains the TIMEOUT value in seconds. If an SQL statement fails, the DBMS will send a message when the timeout period specified by this variable expires. This feature can be useful when searching large databases.
2. The default value of the *option* variable is 120 seconds and the minimum value is 0.
3. Vous pouvez utiliser NS_FUNCTION CHANGEOPTION pour changer la taille maximale du texte (ou de l'image) récupérée par la base de données.

```
;use ns_function CHANGEOPTION to change the
;maxsizes of text (or image) retrieved from
;database

value% =  2147483647 ; max value !

option$ = "TEXTLIMIT"
SQL_EXEC NS_FUNCTION CHANGEOPTION :option$, :value%
if (sql_error% ...)
```

4. Pour connaître, les autres valeurs possibles pour *parameter* reportez-vous à votre manuel de bases de données DB2x.

Example

```
LOCAL TMPS%
LOCAL PARAMETRE$

MOVE 30        TO TMPS%
MOVE "TIMEOUT" TO PARAMETRE$

SQL_EXEC NS_FUNCTION CHANGEOPTION :PARAMETRE$, :TMPS%

SQL_EXEC SELECT * FROM TABLE
IF SQL_ERROR% <> 0
      MESSAGE "ERREUR" , SQL_ERRMSG$(SQL_ERROR%)
      ; you are here if the query takes longer than 30 seconds
      ; to execute
ENDIF
```

NS_FUNCTION CHARTOHEXAOFF, CHARTOHEXAON

CHARTOHEXAON mode automatically converts binary values in STRING or CSTRING form into BINARY or VARBINARY types.

CHARTOHEXAOFF disables automatic conversions.

Syntax **NS_FUNCTION CHARTOHEXAOFF**

and

NS_FUNCTION CHARTOHEXAON

Notes

1. CHARTOHEXAOFF is the active mode by default.
2. In the CHARTOHEXAON mode, BINARY or VARBINARY types are recognized by analyzing the contents of STRING or CSTRING parameters. If the character string begins with 0x, then it is considered to contain a binary value.
3. Constraints on the contents of STRING or CSTRING variables:
   - The first two characters must be "0x" or "0X".
   - The remaining characters must be in the range [0-9] or [A-F].
   - The number of characters must be even.

4. ☞ In CHARTOHEXAON mode, the insertion in the column of the following string ‘0x11FF’ is truncated : ‘0x11’.

Example

```
SQL_EXEC NS_FUNCTION CHARTOHEXAON
...
MOVE '0x11FF' TO A$
SELECT * FROM TABLE WHERE col1 = :A$
; is considered as SELECT * FROM TABLE WHERE col1 = 0x11FF

SQL_EXEC NS_FUNCTION CHARTOHEXAOFF
; Back to the default mode
```

See also SQL_ERROR%, SQL_ERRMSG$

NS_FUNCTION ERRORCOUNT

Retrieves the number of errors or error messages encountered while executing a query. Message numbers start from 0.

Syntax **NS_FUNCTION ERRORCOUNT INTO** *:nbr-errors*

Parameter	*nbr-errors*	INT(4)	I/O	number of errors encountered while executing a query

Example

```
LOCAL I%, ROW_COUNT%, ERROR%
local var1%
local  TOTO$
; the error stack is initialized after every query

var1% = 25
TOTO$ = 3.5

SQL_EXEC  drop table TOTO
SQL_EXEC  INSERT INTO TOTO (NUM, COL1) VALUES (:var1%, :TOTO$)

MOVE 0 TO ROW_COUNT%
SQL_EXEC NS_FUNCTION ERRORCOUNT INTO :ROW_COUNT%
INSERT AT END "ROW_COUNT%" && ROW_COUNT% TO LISTBOX1
;Retrieve the number of errors in ROW_COUNT%
IF ROW_COUNT% <> 0
MOVE 0 TO I%
WHILE i% < ROW_COUNT%
 SQL_EXEC NS_FUNCTION GETERROR :i% INTO :ERROR%
 ; Retrieve for each error the number in ERROR%
 INSERT AT END "ERROR" && I% && SQL_ERRMSG$(ERROR%) TO LISTBOX1
 I% = I% + 1
ENDWHILE
ENDIF
; here the INSERT generate only one error
```

See also NS_FUNCTION GETERROR, SQL_ERROR%, SQL_ERRMSG$, NS_FUNCTION CALLBACK

NS_FUNCTION GETCOLUMN

Retrieves the list of columns for a specified object or for all the objects in the database.

Syntax **NS_FUNCTION GETCOLUMN** *:object-name, :owner-name, :column-name*

Parameters

object-name	CSTRING	I	object name (table or other)
owner-name	CSTRING	I	name of the object's owner
column-name	CSTRING	I	column name used by *object-name*

Notes

1. The parameters passed to NS_FUNCTION GETCOLUMN can be left blank, in which case it assumes that you require all the columns for all the objects in the database.
2. Data is retrieved by executing a FETCH INTO statement that must contain the following items in the order shown:

`:TABLE_QUALIFIER$(129)`	ID of the object qualifier
`:TABLE_OWNER$(129)`	Owner of the object
`:TABLE_NAME$(129)`	Object name
`:COLUMN_NAME$(129)`	Column name
`:DATA_TYPE%(2)`	Column type
`:TYPE_NAME$(129)`	Name of the column type
`:PRECISION%`	Column precision
`:LENGTH%`	Column length
`:SCALE%(2)`	Column dimension
`:RADIX%(2)`	Column radix
`:NULLABLE%(2)`	0=not null, 1=null, 2= unknown
`:REMARKS$(255)`	Column description

Example

```
LOCAL OBJECT_NAME$, OWNER_NAME$, REF_NAME$
LOCAL TABLE_QUALIFIER$(129), TABLE_OWNER$(129), TABLE_NAME$(129), \
COLUMN_NAME$(129), DATA_TYPE%(2)
LOCAL TYPE_NAME$(129), PRECISION%, LENGTH%, SCALE%(2), RADIX%(2), NULLABLE%(2)\
,REMARKS$
sql_init 'NS02DB26'
```

```
if sql_error% <> 0
  message 'error' , sql_errmsg$(sql_error%)
endif
sql_open 'MyDataB', 'db2user/db2user@sample'
if sql_error% <> 0
  message 'error' , sql_errmsg$(sql_error%)
endif

OBJECT_NAME$ =EF_TABLE ;''
OWNER_NAME$ =EF_OWNER ;'DB2USER'
REF_NAME$ = EF_COLUMN ;''

SQL_EXEC NS_FUNCTION GETCOLUMN :OBJECT_NAME$, :OWNER_NAME$, :REF_NAME$
if sql_error% <> 0
  message 'error GETCOLUMN' , sql_errmsg$(sql_error%)
endif

; Recuperation of data
SQL_EXEC FETCH INTO :TABLE_QUALIFIER$, :TABLE_OWNER$, :TABLE_NAME$, \
:COLUMN_NAME$, :DATA_TYPE%, \
:TYPE_NAME$, :PRECISION%, :LENGTH%, :SCALE%, :RADIX%, :NULLABLE%, :REMARKS$
if sql_error% <> 0
  message 'error FETCH' , sql_errmsg$(sql_error%)
endif
WHILE SQL_ERROR% = 0
; always blank
 INSERT AT END "TABLE_QUALIFIER$"&&TABLE_QUALIFIER$ TO LISTBOX1
 INSERT AT END "TABLE_OWNER$"&&TABLE_OWNER$ TO LISTBOX1
 INSERT AT END "TABLE_NAME$"&&TABLE_NAME$ TO LISTBOX1
 INSERT AT END "COLUMN_NAME$"&&COLUMN_NAME$ TO LISTBOX1
 INSERT AT END "DATA_TYPE%"&&DATA_TYPE% TO LISTBOX1
 INSERT AT END "TYPE_NAME$"&&TYPE_NAME$ TO LISTBOX1
 INSERT AT END "PRECISION%"&&PRECISION% TO LISTBOX1
 INSERT AT END "LENGTH%"&&LENGTH% TO LISTBOX1
 INSERT AT END "SCALE%"&&SCALE% TO LISTBOX1
 INSERT AT END "RADIX%"&&RADIX% TO LISTBOX1
 INSERT AT END "NULLABLE%"&&NULLABLE% TO LISTBOX1
 INSERT AT END "REMARKS$"&&REMARKS$ TO LISTBOX1
 SQL_EXEC FETCH INTO :TABLE_QUALIFIER$, :TABLE_OWNER$, :TABLE_NAME$, \
:COLUMN_NAME$, :DATA_TYPE%, \
  :TYPE_NAME$, :PRECISION%, :LENGTH%, :SCALE%, :RADIX%, :NULLABLE%, :REMARKS$
ENDWHILE
```

See also NS_FUNCTION GETINDEXCOLUMN

NS_FUNCTION GETCURRENTDBCNTX

Returns the logical name of the current database.

Syntax **NS_FUNCTION GETCURRENTDBCNTX INTO** : *logical-DBname*

Parameters	*logical-DBname*	CSTRING O	logical database name

Note

1. *logical-DBname* was passed as the first parameter to the SQL_OPEN statement used to open the database.

Example

```
LOCAL HSqlServer%, hdb2% , dbname$

SETPTR GETSPTR%(SPTR_WAIT%)  ; The mouse pointer is changed to
hdb2% = SQL_INITMULTIPLE% ('NS02DB26')
if sql_error% <> 0
  message 'error' , sql_errmsg$(sql_error%)
endif
sql_open 'MyDataB', 'db2user/db2user@sample'
if sql_error% <> 0
  message 'error' , sql_errmsg$(sql_error%)
endif

HSqlServer% = SQL_INITMULTIPLE% ('NS02MS2K')
if sql_error% <> 0
  message 'error' , sql_errmsg$(sql_error%)
endif
sql_open "pubs", " T_IMAGE/aelta@aelta\ADVW135 "
if sql_error% < 0
  message 'open'&&sql_error% , sql_errmsg$(sql_error%)
endif
sql_use hdb2%
sql_exec NS_FUNCTION GETDBNAME into :dbname$ ;'Sample'; the physical name
if sql_error% <> 0
  message 'error GETDBNAME DB2' , sql_errmsg$(sql_error%)&&sql_error%
endif
sql_exec NS_FUNCTION GETCURRENTDBCNTX into :dbname$ ;'MyDataB'; the logical name
if sql_error% <> 0
  message 'error' , sql_errmsg$(sql_error%)&&sql_error%
endif
```

See also SQL_OPEN, SQL_CLOSE, NS_FUNCTION GETDBNAME

NS_FUNCTION GETDBNAME

Retrieves the physical name of the current database.

Syntax **NS_FUNCTION GETDBNAME INTO** : *physical-DBname*

Parameter	*physical-DBname*	CSTRING	I/O	physical name of the current database

Example

```
LOCAL HSqlServer%, hdb2% , dbname$

SETPTR GETSPTR%(SPTR_WAIT%)  ; The mouse pointer is changed to
hdb2% = SQL_INITMULTIPLE% ('NS02DB26')
if sql_error% <> 0
  message 'error' , sql_errmsg$(sql_error%)
endif
sql_open 'MyDataB', 'db2user/db2user@sample'
if sql_error% <> 0
  message 'error' , sql_errmsg$(sql_error%)
endif

HSqlServer% = SQL_INITMULTIPLE% ('NS02MS2K')
if sql_error% <> 0
  message 'error' , sql_errmsg$(sql_error%)
endif
sql_open "pubs", " T_IMAGE/aelta@aelta\ADVW135 "
if sql_error% < 0
  message 'open'&&sql_error% , sql_errmsg$(sql_error%)
endif
sql_use hdb2%
sql_exec NS_FUNCTION GETDBNAME into :dbname$ ;'Sample'; the physical name
if sql_error% <> 0
  message 'error GETDBNAME DB2' , sql_errmsg$(sql_error%)&&sql_error%
endif
sql_exec NS_FUNCTION GETCURRENTDBCNTX into :dbname$ ;'MyDataB'; the logical name
if sql_error% <> 0
  message 'error' , sql_errmsg$(sql_error%)&&sql_error%
endif
```

See also SQL_OPEN, SQL_CLOSE, NS_FUNCTION CHANGEDBCNTX

NS_FUNCTION GETERROR

Retrieves an error code based on its occurrence in the error list. Error numbers lie between 0 and the value returned by NS_FUNCTION ERRORCOUNT minus one.

Syntax **NS_FUNCTION GETERROR** :*error-index%* **INTO** :*error-nbr%*

Parameters

error-index%	INT(4)	I	index of the error number
error-nbr%	INT(4)	O	error number

Example

```
LOCAL i%,ROW_COUNT%, ERROR%
MOVE 0 TO ROW_COUNT%
;Retrieve the number of errors into row_count%
SQL_EXEC NS_FUNCTION ERRORCOUNT INTO :ROW_COUNT%

IF ROW_COUNT% <> 0
   MOVE 0 TO I%
   WHILE I% < ROW_COUNT%
     SQL_EXEC NS_FUNCTION GETERROR :I% INTO :ERROR%
       ; For each error, retrieve its number into ERROR%
     MESSAGE "ERREUR" && I%,SQL_ERRMSG$(ERROR%)
     MOVE I%+1 TO I%
   ENDWHILE
ENDIF
```

See also NS_FUNCTION ERRORCOUNT, SQL_ERROR%, SQL_ERRMSG$, NS_FUNCTION CALLBACK

NS_FUNCTION GETINDEXCOLUMN

NS_FUNCTION GETINDEXCOLUMN does not work with DB2 databases.

Retrieves the list of index columns for a specified object or for all the objects in the database.

Syntax **NS_FUNCTION GETINDEXCOLUMN** *: object-name, :owner-name*

Parameters

object-name	CSTRING	I	object name (table or other)
owner-name	CSTRING	I	name of the object's owner

Notes

1. The parameters passed to NS_FUNCTION GETINDEXCOLUMN can be left blank, in which case it assumes that you require all the index columns for all the objects in the database.
2. Data is retrieved by executing a FETCH INTO statement that must contain the following items in the order shown:

`:TABLE_QUALIFIER$(129)`	ID of the object qualifier
`:TABLE_OWNER$(129)`	Owner of the object
`:TABLE_NAME$(129)`	Object name
`:NON_UNIQUE%(2)`	1 = non-unique, 0 = unique
`:INDEX_QUALIFIER$(129)`	ID of the index qualifier
`:INDEX_NAME$(129)`	Index name
`:TYPE%(2)`	0=STAT, 1=CLUSTER, 2=HASH, 3=other
`:SEQ_IN_INDEX%(2)`	Column sequence number in the index (starts from 1)
`:COLUMN_NAME$(129)`	Column name
`:COLLATION$(2)`	A = Ascending, D = Descending
`:CARDINALITY%`	Cardinality of the table or index
`:PAGES%`	Number of pages for the index
`:FILTER_CONDITION$(129)`	Filter condition for the index

Example

```
SQL_EXEC NS_FUNCTION GETINDEXCOLUMN :OBJECT_NAME$ , :OWNER_NAME$

; Recuperation of data
SQL_EXEC FETCH INTO :TABLE_QUALIFIER$, :TABLE_OWNER$,     \
                    :TABLE_NAME$, :NON_UNIQUE%,           \
                    :INDEX_QUALIFIER$, :INDEX_NAME$,      \
                    :TYPE%, :SEQ_IN_INDEX%, :COLUMN_NAM$, \
                    :COLLATION$, :CARDINALITY%, :PAGES%,  \
                    :FILTER_CONDITION$
```

See also NS_FUNCTION GETCOLUMN

NS_FUNCTION GETINFO

Returns information about the DB2/CLI drivers.

Syntax **NS_FUNCTION GETINFO** *:infotype , :value*

Parameters

infotype	INT(4)	I	Type of information
value	INT(2)/INT(4)/CSTRING	I	Result buffer

Notes

1. The *value* parameter depends on the *infotype* parameter.
2. See the NSDB2.NCL file for more information about the available *infotype*.

Example

```
; recuperation of the DBMS name
LOCAL INFOTYPE%
LOCAL VALUE$
INFOTYPE% = DB_DB2_SERVER_NAME
SQL_EXEC NS_FUNCTION GETINFO :INFOTYPE%, :VALUE$
IF SQL_ERROR% <> 0
  MESSAGE "Error " && SQL_ERROR% , SQL_ERRMSG$(SQL_ERROR%)
ENDIF
INSERT AT END 'Server Name '&& VALUE$ TO LISTBOX1
INFOTYPE% = DB_DB2_DATABASE_NAME
SQL_EXEC NS_FUNCTION GETINFO :INFOTYPE%, :VALUE$
IF SQL_ERROR% <> 0
  MESSAGE "Error " && SQL_ERROR% , SQL_ERRMSG$(SQL_ERROR%)
ENDIF
INSERT AT END 'DATABASE Name '&& VALUE$ TO LISTBOX1
INFOTYPE% = DB_DB2_DATA_SOURCE_NAME
SQL_EXEC NS_FUNCTION GETINFO :INFOTYPE%, :VALUE$
IF SQL_ERROR% <> 0
  MESSAGE "Error " && SQL_ERROR% , SQL_ERRMSG$(SQL_ERROR%)
ENDIF
INSERT AT END 'DATA SOURCE Name '&& VALUE$ TO LISTBOX1
```

See also SQL_ERROR%, SQL_ERRMSG$

NS_FUNCTION GETPRIMARYKEY

Retrieves the columns that form the primary key for an object or for all the objects in the database.

Syntax **NS_FUNCTION GETPRIMARYKEY** *:object-name, :owner-name*

Parameters

object-name	CSTRING	I	object name (table or other)
owner-name	CSTRING	I	name of the object's owner

Notes

1. The parameters passed to NS_FUNCTION GETPRIMARYKEY can be left blank, in which case it assumes that you require the primary keys for all the objects in the database.
2. Data is retrieved by executing a FETCH INTO statement that must contain the following items:

`:TABLE_QUALIFIER$(129)`	ID of the object qualifier
`:TABLE_OWNER$(129)`	Owner of the object
`:TABLE_NAME$(129)`	Object name
`:COLUMN_NAME$(129)`	Column name
`:KEY_SEQ%(2)`	Column sequence number in the primary key
`:PK_NAME$(129)`	ID of the column qualifier

Example 1

```
SQL_EXEC NS_FUNCTION GETPRIMARYKEY :OBJECT_NAME$ , :OWNER_NAME$

; Recuperation of data
SQL_EXEC FETCH INTO :TABLE_QUALIFIER$, :TABLE_OWNER$, \
                    :TABLE_NAME$, :COLUMN_NAME$,      \
                    :KEY_SEQ%(2), :PK_NAME$
```

Example 2

```
; Creation

sql_exec drop table EMP
if sql_error% <> 0
  message 'error drop' , sql_errmsg$(sql_error%)
endif

SQL_EXEC CREATE TABLE EMP ( \
     EMPNO INTEGER  NOT NULL PRIMARY KEY, \
     ENAME CHAR(25))
if sql_error% <> 0
  message 'error create' , sql_errmsg$(sql_error%)
endif

;then GETPRIMARYKEY

LOCAL OBJECT_NAME$, OWNER_NAME$, REF_NAME$
LOCAL TABLE_QUALIFIER$(129), TABLE_OWNER$(129), TABLE_NAME$(129)
LOCAL  COLUMN_NAME$(129), KEY_SEQ%(2), PK_NAME$(129)
OBJECT_NAME$ ='EMP'
OWNER_NAME$ ='Scott'

SQL_EXEC NS_FUNCTION GETPRIMARYKEY :OBJECT_NAME$, :OWNER_NAME$
if sql_error% <> 0
  message 'error GETPRIMARYKEY' , sql_error%&&sql_errmsg$(sql_error%)
endif

; Recuperation of data
SQL_EXEC FETCH INTO :TABLE_QUALIFIER$, :TABLE_OWNER$, :TABLE_NAME$,
:COLUMN_NAME$, KEY_SEQ%, :PK_NAME$
if sql_error% <> 0
  message 'error FETCH' , sql_error%&&sql_errmsg$(sql_error%)
endif
WHILE SQL_ERROR% = 0
      INSERT AT END "TABLE_QUALIFIER$"&&TABLE_QUALIFIER$ TO LISTBOX1
      INSERT AT END "TABLE_OWNER$"&&TABLE_OWNER$ TO LISTBOX1
      INSERT AT END "TABLE_NAME$"&&TABLE_NAME$ TO LISTBOX1
      INSERT AT END "COLUMN_NAME$"&&COLUMN_NAME$ TO LISTBOX1
      INSERT AT END "KEY_SEQ%"&&KEY_SEQ% TO LISTBOX1
      INSERT AT END "PK_NAME$"&&PK_NAME$ TO LISTBOX1
      SQL_EXEC FETCH INTO :TABLE_QUALIFIER$, :TABLE_OWNER$, :TABLE_NAME$,
:COLUMN_NAME$, KEY_SEQ%, :PK_NAME$
ENDWHILE

; and then we get the key in COLUMN_NAME$
```

See also NS_FUNCTION GETCOLUMN, NS_FUNCTION GETINDEXCOLUMN

NS_FUNCTION GETPROCEDURE

Retrieves the list of stored procedures and functions, if the DATA SOURCE supports these features.

Syntax **NS_FUNCTION GETPROCEDURE**

Notes

1. Data is retrieved by executing a FETCH INTO statement that must contain the following items in the order shown:

`:PROCEDURE_QUALIFIER$(129)`	ID of the procedure qualifier
`:PROCEDURE_OWNER$(129)`	Owner of the procedure
`:PROCEDURE_NAME$(129)`	Procedure name
`:NUM_INPUT_PARAMS%`	Reserved
`:NUM_OUPUT_PARAM%`	Reserved
`:NUM_RESULT_SETS%`	Reserved
`:REMARK$(129)`	Description of the procedure
`:PROCEDURE_TYPE%(2)`	0 = unknown, 1 =procedure, 2=function

Example

```
SQL_EXEC NS_FUNCTION GETPROCEDURE

; recuperation of data
SQL_EXEC FETCH INTO :PROCEDURE_QUALIFIER$,:PROCEDURE _OWNER$,\
                    :PROCEDURE _NAME$, :NUM_INPUT_PARAMS%,   \
                    :NUM_OUPUT_PARAM%, :NUM_RESULT_SETS%,    \
                    :REMARK$, :PROCEDURE_TYPE%
```

See also NS_FUNCTION GETPROCEDURECOLUMN

NS_FUNCTION GETPROCEDURECOLUMN

Retrieves the list of parameters for a specified procedure or for all procedures.

Syntax **NS_FUNCTION GETPROCEDURECOLUMN** *:object-name, :owner-name, :parameter-name*

Parameters

object-name	CSTRING	I	procedure name
owner-name	CSTRING	I	owner name
parameter-name	CSTRING	I	parameter name used by *object-name*

Notes

1. The parameters passed to NS_FUNCTION GETPROCEDURECOLUMN can be left blank, in which case it assumes that you require all the parameters for all the procedures in the database.
2. Data is retrieved by executing a FETCH INTO statement that must contain the following items:

`:PROCEDURE_QUALIFIER$(129)`	ID of the procedure qualifier
`:PROCEDURE _OWNER$(129)`	Owner of the procedure
`:PROCEDURE _NAME$(129)`	Procedure name
`:COLUMN_NAME$(129)`	Procedure parameter name
`:COLUMN_TYPE%(2)`	0 = unknown, 1 = input parameter 2 = input/output parameter, 3 = output parameter, 4 = return code
`:DATA_TYPE%(2)`	Parameter type
`:TYPE_NAME$(129)`	Type name
`:PRECISION%`	Parameter precision
`:LENGTH%`	Parameter length
`:SCALE%(2)`	Parameter's dimension
`:RADIX%(2)`	Parameter's radix
`:NULLABLE%(2)`	0 = not null, 1 = null, 2 = unknown
`:REMARK$(255)`	Parameter description

Example

```
SQL_EXEC NS_FUNCTION GETPROCEDURECOLUMN :OBJECT_NAME$ , \
                                        :OWNER_NAME$ ,  \
                                        :REF_NAME$\
SQL_EXEC FETCH INTO ... ; recuperation of data
```

See also NS_FUNCTION GETPROCEDURE

NS_FUNCTION GETTABLE

Returns the list of objects of a given type.

Syntax **NS_FUNCTION GETTABLE** *: object-type, :owner-name*

Parameters

object-type	CSTRING	I	object type (TABLE, SYSTEM TABLE, etc.)
owner-name	CSTRING	I	owner name

Notes

1. The parameters passed to NS_FUNCTION GETTABLE can be left blank, in which case it assumes that you require all the objects in the database for all types.
2. Data is retrieved by executing a FETCH INTO statement that must contain the following items:

`:TABLE_QUALIFIER$(129)`	ID of the object qualifier
`:TABLE_OWNER$(129)`	Owner of the object
`:TABLE_NAME$(129)`	Object name
`:TABLE_TYPE$(129)`	TABLE, VIEW, SYSTEM TABLE, GLOBAL TEMPORARY, LOCAL TEMPORARY, ALIAS, SYNONYM
`:REMARKS$(255)`	Object description

Example

```
LOCAL OBJECT_NAME$, OWNER_NAME$, REF_NAME$
LOCAL TABLE_QUALIFIER$(129), TABLE_OWNER$(129), TABLE_NAME$(129)
LOCAL TABLE_TYPE$(129), REMARKS$
OBJECT_NAME$ =EF_TABLE
OWNER_NAME$ =EF_OWNER

SQL_EXEC NS_FUNCTION GETTABLE :OBJECT_NAME$, :OWNER_NAME$
if sql_error% <> 0
  message 'error GETTABLE' , sql_errmsg$(sql_error%)
endif
```

```
; Recuperation of data
SQL_EXEC FETCH INTO :TABLE_QUALIFIER$, :TABLE_OWNER$, :TABLE_NAME$,
:TABLE_TYPE$, :REMARKS$
if sql_error% <> 0
  message 'error FETCH' , sql_errmsg$(sql_error%)
endif
WHILE SQL_ERROR% = 0
 INSERT AT END "TABLE_QUALIFIER$"&&TABLE_QUALIFIER$ TO LISTBOX1
 INSERT AT END "TABLE_OWNER$"&&TABLE_OWNER$ TO LISTBOX1
 INSERT AT END "TABLE_NAME$"&&TABLE_NAME$ TO LISTBOX1
 INSERT AT END "TABLE_TYPE$"&&TABLE_TYPE$ TO LISTBOX1
 INSERT AT END "REMARKS$"&&REMARKS$ TO LISTBOX1
 SQL_EXEC FETCH INTO :TABLE_QUALIFIER$, :TABLE_OWNER$, :TABLE_NAME$, \
:TABLE_TYPE$, :REMARKS$
ENDWHILE
```

See also NS_FUNCTION GETPROCEDURECOLUMN

NS_FUNCTION GETTABLEINFO

Returns the description of a given object.

Syntax **NS_FUNCTION GETTABLEINFO** *:object-type, :owner-name, :object-name*

Parameters

object-type	CSTRING	I	object type (TABLE, SYSTEM TABLE, etc.)
owner-name	CSTRING	I	owner name
object-name	CSTRING	I	object name

Notes

1. The parameters passed to NS_FUNCTION GETTABLE can be left blank
2. Data is retrieved by executing a FETCH INTO statement that must contain the following items:

`:TABLE_QUALIFIER$(129)`	ID of the object qualifier
`:TABLE_OWNER$(129)`	Owner of the object
`:TABLE_NAME$(129)`	Object name
`:TABLE_TYPE$(129)`	TABLE, VIEW, SYSTEM TABLE, GLOBAL TEMPORARY, LOCAL TEMPORARY, ALIAS, SYNONYM
`:REMARKS$(255)`	Object description

Example

```
LOCAL OBJECT_NAME$, OWNER_NAME$, REF_NAME$
LOCAL TABLE_QUALIFIER$(129), TABLE_OWNER$(129), TABLE_NAME$(129)
LOCAL TABLE_TYPE$(129),  REMARKS$
OBJECT_NAME$ =EF_TABLE
OWNER_NAME$ =EF_OWNER
REF_NAME$ = EF_COLUMN


SQL_EXEC NS_FUNCTION GETTABLEINFO :OBJECT_NAME$, :OWNER_NAME$, :REF_NAME$
if sql_error% <> 0
  message 'error GETTABLEINFO' , sql_errmsg$(sql_error%)
endif

; Recuperation of data
```

```
SQL_EXEC FETCH INTO :TABLE_QUALIFIER$, :TABLE_OWNER$, :TABLE_NAME$, \
:TABLE_TYPE$, :REMARKS$
if sql_error% <> 0
  message 'error FETCH' , sql_errmsg$(sql_error%)
endif
WHILE SQL_ERROR% = 0
; always blank
; INSERT AT END "TABLE_QUALIFIER$"&&TABLE_QUALIFIER$ TO LISTBOX1
 INSERT AT END "TABLE_OWNER$"&&TABLE_OWNER$ TO LISTBOX1
 INSERT AT END "TABLE_NAME$"&&TABLE_NAME$ TO LISTBOX1
 INSERT AT END "TABLE_TYPE$"&&TABLE_TYPE$ TO LISTBOX1
 INSERT AT END "REMARKS$"&&REMARKS$ TO LISTBOX1
 SQL_EXEC FETCH INTO :TABLE_QUALIFIER$, :TABLE_OWNER$, :TABLE_NAME$, \
:TABLE_TYPE$, :REMARKS$
ENDWHILE
```

See also NS_FUNCTION GETTABLE

NS_FUNCTION GETTYPEINFO

Retrieves the SQL data type.

Syntax	**NS_FUNCTION GETTYPEINFO** : *object-type*			
Parameter	*object-type*	INT(4)	I	SQL data type

Note

1. SQL data types can have the following values:

0	all data types
1	BIGINT
2	BINARY
3	BIT
4	CHAR
5	DATE
6	DECIMAL
7	DOUBLE
8	FLOAT
9	INTEGER
10	LONGVARBINARY
11	LONGVARCHAR
12	LONGVARBINARY
13	NUMERIC
14	REAL
15	SMALLINT
16	TIME
17	TIMESTAMP
18	TINYINT
19	VARBINARY
20	VARCHAR

2. Data is retrieved by executing a FETCH INTO statement that must contain the following items:

`:TYPE_NAME$(129)`	Data type name used by the DATASOURCE
`:DATA_TYPE%(2)`	SQL data type (DB2/CLI format)
`:PRECISION%(4)`	Maximum precision value
`:LITERAL_PREFIX$(128)`	Character (s) used to prefix a literal value

	(e.g. "" for CHAR)
`:LITERAL_SUFFIX$(128)`	Character (s) used to suffix a literal value
	(e.g. "" for CHAR)
`:CREATE_PARAMS$(128)`	List of parameters for this data type
`:NULLABLE%(2)`	0 = NULL not allowed, 1 = NULL allowed
`:CASE_SENSITIVE%(2)`	1 = case sensitive, 0 = non case sensitive
`:SEARCHABLE%(2)`	0 = data type incompatible with WHERE
	1 = data type only compatible with LIKE
	2 = data type incompatible with LIKE
	3 = data type fully compatible with WHERE
`:UNSIGNED_ATTRIBUTE%(2)`	0 = data type does not support + or - sign
	1 = data type supports + or - sign
`:MONEY%(2)`	0 = non-money data type, 1 = money data type
`:AUTO_INCREMENT%(2)`	0 = no auto increment,
	1 = supports auto increment
`:LOCAL_TYPE_NAME$(128)`	Data type name in SQL engine
`:MINIMUM_SCALE%(2)`	Minimum precision value
`:MAXIMUM_SCALE%(2)`	Maximum precision value

Example

```
LOCAL OBJECT_TYPE%
LOCAL TYPE_NAME$(129), DATA_TYPE%(2), PRECISION%
LOCAL LITERAL_PREFIX$(128), LITERAL_SUFFIX$(128), CREATE_PARAMS$(128)
LOCAL NULLABLE%(2), CASE_SENSITIVE%(2), SEARCHABLE%(2), UNSIGNED_ATTRIBUTE%(2)
LOCAL MONEY%(2), AUTO_INCREMENT%(2)
LOCAL LOCAL_TYPE_NAME$(128), MINIMUM_SCALE%(2), MAXIMUM_SCALE%(2)
OBJECT_TYPE% =EF_TYPE

SQL_EXEC NS_FUNCTION GETTYPEINFO :OBJECT_TYPE%
if sql_error% <> 0
  message 'error GETTYPEINFO' , sql_errmsg$(sql_error%)
endif

; Récupération des données
SQL_EXEC FETCH INTO :TYPE_NAME$, :DATA_TYPE%, :PRECISION%, :LITERAL_PREFIX$, \
:LITERAL_SUFFIX$, :CREATE_PARAMS$, \
 :NULLABLE%, :CASE_SENSITIVE%, :SEARCHABLE%, :UNSIGNED_ATTRIBUTE%, :MONEY%, \
 :AUTO_INCREMENT% ,  \
 :LOCAL_TYPE_NAME$, :MINIMUM_SCALE%, :MAXIMUM_SCALE%
if sql_error% <> 0
  message 'error FETCH' , sql_errmsg$(sql_error%)
endif
WHILE SQL_ERROR% = 0
 INSERT AT END "TYPE_NAME$"&&TYPE_NAME$ TO LISTBOX1
 INSERT AT END "DATA_TYPE%"&&DATA_TYPE% TO LISTBOX1
 INSERT AT END "PRECISION%"&&PRECISION% TO LISTBOX1
 INSERT AT END "LITERAL_PREFIX$"&&LITERAL_PREFIX$ TO LISTBOX1
 INSERT AT END "LITERAL_SUFFIX$"&&LITERAL_SUFFIX$ TO LISTBOX1
 INSERT AT END "CREATE_PARAMS$"&&CREATE_PARAMS$ TO LISTBOX1
 INSERT AT END "NULLABLE%"&&NULLABLE% TO LISTBOX1
 INSERT AT END "CASE_SENSITIVE%,"&&CASE_SENSITIVE% TO LISTBOX1
 INSERT AT END "SEARCHABLE%"&&SEARCHABLE% TO LISTBOX1
 INSERT AT END "UNSIGNED_ATTRIBUTE%"&&UNSIGNED_ATTRIBUTE% TO LISTBOX1
 INSERT AT END "MONEY%"&&MONEY% TO LISTBOX1
 INSERT AT END "AUTO_INCREMENT%"&&AUTO_INCREMENT% TO LISTBOX1
 INSERT AT END "LOCAL_TYPE_NAME$"&&LOCAL_TYPE_NAME$ TO LISTBOX1
 INSERT AT END "MINIMUM_SCALE%"&&MINIMUM_SCALE% TO LISTBOX1
 INSERT AT END "MAXIMUM_SCALE%"&&MAXIMUM_SCALE% TO LISTBOX1
SQL_EXEC FETCH INTO :TYPE_NAME$, :DATA_TYPE%, :PRECISION%, :LITERAL_PREFIX$,
:LITERAL_SUFFIX$, :CREATE_PARAMS$, \
 :NULLABLE%, :CASE_SENSITIVE%, :SEARCHABLE%, :UNSIGNED_ATTRIBUTE%, :MONEY%,
:AUTO_INCREMENT% \
 :LOCAL_TYPE_NAME$, :MINIMUM_SCALE%, :MAXIMUM_SCALE%
ENDWHILE
```

See also SQL_ERROR%, SQL_ERRMSG$

NS_FUNCTION GIVECOM

Retrieves in COM_AREA segment the characteristics of a table whose components are unknown when a SELECT statement is executed.

This function is particularly useful in the treatment of dynamic requests and avoid to define host variables and FETCH command.

Syntax **NS_FUNCTION GIVECOM INTO** : *table-characteristics*

Variables

table-characteristics	INT(4)	I/O	pointer to the segment COM_AREA used to retrieve the table's characteristics

Notes

1. The selection structure is retrieved in a segment COM_AREA (defined in the SQL_COM.NCL file). The host variable *table-characteristics* contains the pointer to this segment.

2. This host segment comprises a number of field, including two pointers (HOST_PTR and SQL_PTR). These pointers can be retrieved in order to scan the two arrays that respectively contain the NCL variables and SQL variables affected by the query about to be executed.

```
; Definition of the communication structure (GIVECOM INTO:)
SEGMENT COM_AREA
    int reserved(4)     ;reserved
    int transaction(2)  ;reserved
    int statement(2)    ;reserved
    int host_ptr(4)     ;pointer to a segment of a NCLELEMENT type
                        ;(defining the NCL host variables)
    int sql_ptr(4)      ;pointer to a segment of a SQLELEMENT type
                        ;(defining the columns of the request tables)
    int com_ptr(4)      ;reserved
    int num_stat(2)     ;type of the request
                        ; 1 -> SELECT
                        ; 2 -> UPDATE
                        ; 3 -> DELETE
                        ; 4 -> INSERT
                        ; 5 -> Other
    int num_col(2)      ; number of the columns
    int num_col_compute(2) ; number of the COMPUTE columns (not
                           ; applicable for Oracle)
    int len_buf_stat(2) ; size of the buf_stat below
    int buf_stat(4)     ; pointer on a buffer containing
;the FETCH INTO [ :,] instruction followed by « :, » as much as
;variables to go through in a SELECT case
    int inited(2)       ;TRUE if all is OK, FALSE otherwise.
                        ;Always to test if it’s TRUE.
ENDSEGMENT
```

3. The SQL_COM.NCL library provides a set of functions required to make use of the NS_FUNCTION GIVECOM INTO function:

- Communication structure.
- Functions that return the type of command to be executed.
- All the functions used to retrieve pointers.
- Types, sizes and names of the columns affected by the selection.

4. Once the type of the command has been identified as a SELECT statement (after using the SQL_GET_STATEMENT% and SQL_GET_STATEMENT$ functions), the SQL_EXEC_LONGSTR command can execute the query that will fill the receiving field. The results can be extracted from this field using the functions in the NCL library.

5. The following is a list of functions in the NCL library:

- **FUNCTION SQL_GET_HOSTPTR%**

  Returns a pointer to an array of variables named COM_NCLELEMENT (definition of NCL host variables).

Variable	*COM_BUFFER%*	INT(4)	Pointer to COM_AREA

  Return value INT(4)

```
; Definition of the receiver NCL variables’s structure
SEGMENT COM_NCLELEMENT
    int buffer_ptr(4)
    int ncltype(2)
    integer ncllength
    int reserved1(2)
    int reserved2(2)
ENDSEGMENT
```

- **FUNCTION SQL_GET_SQLPTR%**

  Returns a pointer to an array of variables named COM_SQLELEMENT.

Variable	*COM_BUFFER%*	INT(4)	Pointer to COM_AREA

  Return value INT(4)

```
; Definition of the SQL column’s structure
SEGMENT COM_SQLELEMENT
    CSTRING colname(64)             ; name of the column
    int collength(4)                ; size of the column
    int coltype(2)                  ; type of the column
    int colservice(2)               ; service for this column
    int colcomputeref(2)            ; reference of the column having the compute
ENDSEGMENT
```

- **FUNCTION SQL_GET_STATEMENT%**

Returns the type of statement executed (integer value) from the num_stat buffer of COM_AREA segment.

Variable	*COM_BUFFER%*	INT(4)	Pointer to COM_AREA

Return value INT(2)

- **FUNCTION SQL_GET_STATEMENT$**

Returns the type of statement executed (alphanumeric value) from num_stat buffer of COM_AREA segment and converts it to a CSTRING value.

Variable	*STATEMENT%*	INT(4)	SQL_GET_STATEMENT% function

Return value CSTRING

The values of num_stat are the following :
- 1 for SELECT
- 2 for UPDATE
- 3 for DELETE
- 4 for INSERT
- 0 for any other type of request.

- **FUNCTION SQL_GET_NBCOL%**

Returns the number of columns retrieved by the statement.

Variable	*COM_BUFFER%*	INT(4)	Pointer to COM_AREA

Return value INT(2)

- **FUNCTION SQL_GET_LENGTHFETCH%**

Returns the size of the fetch buffer..

Variable	*COM_BUFFER%*	INT(4)	Pointer to COM_AREA

Return value INT(4)

- **FUNCTION SQL_GET_FETCHPTR%**

Returns the pointer to the fetch buffer.

Variable	*COM_BUFFER%*	INT(4)	Pointer to COM_AREA

Return value INT(4)

- **FUNCTION SQL_GET_HOSTCOLUMNPTR%**

Returns the pointer to the data in an element in the array of NCL variables.

Variable	*COM_BUFFER%*	INT(4)	Pointer to COM_AREA
	COLUMN%	INT(2)	Order of the NCL variable

Return value INT(4)

- **FUNCTION SQL_GET_HOSTCOLUMNTYPE%**

Returns the data type for an element in the array of NCL variables (integer value).

Variable	*COM_BUFFER%*	INT(4)	Pointer to COM_AREA
	COLUMN%	INT(2)	Order of the NCL variable

Return value INT(2)

- **FUNCTION SQL_GET_HOSTCOLUMNTYPE$**

Returns the data type for an element in the array of NCL variables (alphanumeric value).

Variable	*TYPE%*	INT(4)	SQL_GET_HOSTCOLUMNLENGTH% function

Return value CSTRING(80)

- **FUNCTION SQL_GET_HOSTCOLUMNLENGTH%**

Returns the data size for an element in the array of NCL variables.

Variable	*COM_BUFFER%*	INT(4)	Pointer to COM_AREA
	COLUMN%	INT(2)	Order of the NCL variable

Return value INT(2)

- **FUNCTION SQL_GET_SQLCOLUMNNAME$**

Returns the column name in the array of SQL columns.

Variable	*COM_BUFFER%*	INT(4)	Pointer to COM_AREA
	COLUMN%	INT(2)	Order of the NCL variable

Return value CSTRING(64)

The five following functions are not very useful with NS_FUNCTION GIVECOM. However, we let's them in the documentation to compatibility with older documentations.

- **FUNCTION SQL_GET_SQLCOLUMNTYPE%**

Returns the DBMS column type in the array of SQL columns.

```
FUNCTION SQL_GET_SQLCOLUMNTYPE% \
    (INT COM_BUFFER%(4),INT COLUMN%(2))\
         RETURN INT(2)
```

- **FUNCTION SQL_GET_SQLCOLUMNLENGTH%**

Returns the DBMS column size in the array of SQL columns.

```
FUNCTION SQL_GET_SQLCOLUMNLENGTH% \
      (INT COM_BUFFER%(4),INT COLUMN%(2))\
         RETURN INT(4)
```

- **FUNCTION SQL_GET_SQLCOLUMNSERVICE%**

Retrieves the DBMS column service in the array of SQL columns (integer value).

```
FUNCTION SQL_GET_SQLCOLUMNSERVICE%  \
(INT COM_BUFFER%(4),INT COLUMN%(2)) \
         RETURN INT(2)
```

- **FUNCTION SQL_GET_SQLCOLUMNREF%**

Retrieves the column number referenced by COMPUTE.

```
FUNCTION SQL_GET_SQLCOLUMNREF% \
  (INT COM_BUFFER%(4),INT COLUMN%(2))  \
         RETURN INT(2)
```

- **FUNCTION SQL_GET_SQLCOLUMNSERVICE$**

Retrieves the DBMS service (alphanumeric value).

```
FUNCTION SQL_GET_SQLCOLUMNSERVICE$ \
   (INT service%(2)) \
         RETURN CSTRING(80)
```

Example

```
LOCAL COM_AREA_RET%, TOTAL_COL%, I%, NCL_PTR%, RET%, BUFFER_PTR%
LOCAL  COMPUTE% , HEADER$
```

```
MOVE "SELECT * FROM AUTHORS" TO A$
SQL_EXECSTR A$

MOVE 0 TO RET%
MOVE 0 TO COM_AREA_RET%

WHILE RET% = 0
      SQL_EXEC NS_FUNCTION GIVECOM INTO:COM_AREA_RET%

      IF COM_AREA_RET% = 0
             BREAK
      ENDIF
      IF SQL_GET_STATEMENT%(COM_AREA_RET%) <> 1
             ; if the statement is not SELECT
             ; retrieve the statement string
             INSERT AT END SQL_GET_STATEMENT$ \
             (SQL_GET_STATEMENT%(COM_AREA_RET%) ) TO LIST_FETCH
              UPDATE LIST_FETCH
             RETURN 1
      ENDIF

      MOVE SQL_GET_HOSTPTR%(COM_AREA_RET%) TO NCL_PTR%
      ; retrieve the pointer to the array of NCL variables
      MOVE SQL_GET_NBCOL%(COM_AREA_RET%) + SQL_GET_NBCOMPUTE%(COM_AREA_RET%) \
             TO TOTAL_COL%
      ; retrieve the number of columns  + the number of COMPUTE columns
      IF SQL_GET_LENGTHFETCH%(COM_AREA_RET%) <> 0
             ; if the Fetch buffer size <> 0
             SQL_EXEC_LONGSTR (SQL_GET_FETCHPTR%(COM_AREA_RET%) , NCL_PTR%, -1)
             ; retrieve the pointer to the Fetch buffer + execute
        ELSE
             BREAK
        ENDIF
        WHILE RET% = 0 OR RET% = 200
          ; +200 message returned by SELECT...COMPUTE
          IF RET% = 200
             MOVE 1 TO COMPUTE%
          ELSE
             MOVE 0 TO COMPUTE%
          ENDIF
          MOVE 0 TO I%
          WHILE I% < TOTAL_COL%
             MOVE SQL_GET_HOSTCOLUMNPTR%(COM_AREA_RET% ,i% ) TO BUFFER_PTR%
             ; retrieve a pointer to an NCL variable
             IF BUFFER_PTR% = 0
               MOVE I% + 1 TO I%
               CONTINUE
             ENDIF
             IF COMPUTE% = 1
               ; assign SQL table column name
               MOVE "COMPUTE OF " && SQL_GET_SQLCOLUMNNAME$(COM_AREA_RET% ,\
               SQL_GET_SQLCOLUMNREF% (COM_AREA_RET% ,i% ) -1 ) TO HEADER$
               ; store the column name retrieved from the SQL array
               IF SQL_GET_SQLCOLUMNSERVICE% (COM_AREA_RET% ,i%) <> 3
                    ; if the service is not COMPUTE
                    MOVE I% + 1 TO I%
                    CONTINUE
               ENDIF

                 ELSE
                    MOVE "" to HEADER$
                    IF SQL_GET_SQLCOLUMNSERVICE% (COM_AREA_RET%,i%) <> 1
                    ; if the service is not a column
                    MOVE I% + 1 TO I%
                    CONTINUE
                 ENDIF
             ENDIF
             EVALUATE SQL_GET_HOSTCOLUMNTYPE%(COM_AREA_RET% ,i% )
             ; determine column type
               WHERE 0 ; integer
                 ; retrieve the size of the data item
                 EVALUATE SQL_GET_HOSTCOLUMNLENGTH%(COM_AREA_RET% ,i%)
                   WHERE 1 ; 1-byte integer
                    ; retrieve column name
                    INSERT AT END HEADER$ && SQL_GET_SQLCOLUMNNAME$\
                    (COM_AREA_RET%, I% ) && ":" && ASC% \
                    (COM_INT1(BUFFER_PTR%).i1) TO LIST_FETCH
                   ENDWHERE
                   WHERE 2 ; 2-byte integer
                    ; retrieve column name
                    INSERT AT END HEADER$ && \
                    SQL_GET_SQLCOLUMNNAME$(COM_AREA_RET% ,i% )&&\
                           ":" && COM_INT2(BUFFER_PTR%).i2  TO LIST_FETCH
                   ENDWHERE
                   WHERE 4 ; 4-byte integer
                    INSERT AT END HEADER$ && \
                    SQL_GET_SQLCOLUMNNAME$(COM_AREA_RET% ,i% )&&\
                    ":" && COM_INT4(BUFFER_PTR%).i4 TO LIST_FETCH
                   ENDWHERE
                 ENDEVALUATE
               ENDWHERE
               WHERE 2 ; C string
                    INSERT AT END HEADER$ && \
                    SQL_GET_SQLCOLUMNNAME$(COM_AREA_RET%, I% ) &&\
                      ":" && COM_STRING(BUFFER_PTR%).CS TO LIST_FETCH
               ENDWHERE
               WHERE 3 ; real
                    EVALUATE SQL_GET_HOSTCOLUMNLENGTH%(COM_AREA_RET%, I%)
                      ; retrieve column size
                      WHERE 4 ;
                           ; retrieve column name and real 4-byte value
                           INSERT AT END HEADER$ && \
                           SQL_GET_SQLCOLUMNNAME$(COM_AREA_RET%, I% ) &&\
                             ":" && COM_FLOAT4(BUFFER_PTR%).f4 TO LIST_FETCH
                      ENDWHERE
                      WHERE 8 ;
                           ; retrieve column name and 8-byte real value
                           INSERT AT END HEADER$ && \
                           SQL_GET_SQLCOLUMNNAME$(COM_AREA_RET%, I% ) &&\
                           ":" && COM_FLOAT8(BUFFER_PTR%).f8 TO LIST_FETCH
                      ENDWHERE
                    ENDEVALUATE
               ENDWHERE
               ELSE
               ; retrieve type of NCL column
               INSERT AT END "NCLType" && "INVALID" && \
               SQL_GET_HOSTCOLUMNTYPE%(COM_AREA_RET%, i%) TO LIST_FETCH
               ; retrieve NCL column type
             ENDEVALUATE
             MOVE I% + 1 to I%
          ENDWHILE

          SQL_EXEC_LONGSTR SQL_GET_FETCHPTR%(COM_AREA_RET%),NCL_PTR%,-1
          ; execute FETCH using pointer to FETCH buffer

      ENDWHILE

      UPDATE LIST_FETCH
      IF RET% = 100
             INSERT AT END "END OF FETCH" TO LIST_FETCH
             INSERT AT END "" TO LIST_FETCH
             MOVE 0 TO RET%
      ENDIF
      IF RET% <> 100
             IF RET% > 0
                    MESSAGE "WARNING" && RET% , SQL_ERRMSG$(RET%)
             ENDIF
             IF RET% < 0
                    IF RET% = -32085
                           INSERT AT END "END OF RESULT" TO LIST_FETCH
                    ELSE
                           MESSAGE "ERROR" && RET% , SQL_ERRMSG$(RET%)
                    ENDIF
             ENDIF
      ENDIF

ENDWHILE
```

See also SQL_EXEC_LONGSTR.

NS_FUNCTION IMAGEOFF, IMAGEON

IMAGEON mode enables binary object management (for example bitmaps) of 32000 bytes maximum size. This manipulation has been executed in NCL segment SQL_IMAGE defined in NSDMS.NCL.

IMAGEOFF mode inactivates this function.

Syntax

NS_FUNCTION IMAGEOFF

and

NS_FUNCTION IMAGEON

Notes

1. IMAGEOFF is the mode by default.
2. Binary objects are manipulated using an NCL segment SQL_IMAGE:

```
SEGMENT SQL_IMAGE
 INT REALSIZE(4) ; size of buffer allocated
 INT LENGTH%(4)  ; real size
                 ; (from SELECT)
  INT PTR%(4)    ; Buffer address
ENDSEGMENT
```

3. The maximum authorized size is 32K. If you want to handle BLOBs (large images) see TYPE_SQL_INSERT_BLOB% and TYPE_SQL_SELECT_BLOB%.
4. Images are not the only type of binary objects. Any type of binary file can be stored.
5. Binary storage is not cross-platform. Therefore, if you store binary files using Windows (ANSI) and you want to retrieve it using OS/2, you will probably have problems.

Example

```
;creation of a table

SQL_EXEC CREATE TABLE T_IMAGE(NUMERO NUMBER(8), DESCRIPTION VARCHAR2(80)\
      , IMAGE LONG RAW)
if sql_error% <> 0
  message 'error Create' , sql_errmsg$(sql_error%)
endif

;INSERT

LOCAL DEST$(80),DATA%,SIZE%(4),NBREAD%(2),FILE%,NIL%,FNAME$, hbmp%
LOCAL SQL_IMAGE LOCALIMAGE

SQL_EXEC NS_FUNCTION IMAGEON
IF SQL_ERROR% <> 0
 MESSAGE "IMAGEON", SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
 RETURN 1
ENDIF
FNAME$ = "(NS-BMP)\TINTIN.BMP"
HBMP%=CREATEBMP%(FNAME$)
BMPT = HBMP%
;FGETSIZE% does not accept  environnement variables
FNAME$ = "D:\TESTS\BMP\TINTIN.BMP"

SIZE%=FGETSIZE%(FNAME$) INSERT AT END "SIZE "& SIZE% TO LISTBOX1
NEW SIZE%,DATA%
FILE%=F_OPEN%(1,FNAME$)
F_BLOCKREAD FILE%, DATA%, SIZE%, NBREAD%
IF F_ERROR%
 MESSAGE"ERROR", "Failed to load " & FNAME$ &"!"
 F_CLOSE FILE%
 DISPOSE DATA%
 RETURN 1
ENDIF
; ---- Insert into table t_image
LOCALIMAGE.REALSIZE = SIZE%
LOCALIMAGE.LENGTH% = SIZE%
LOCALIMAGE.PTR% = DATA%
SQL_EXEC INSERT INTO T_IMAGE VALUES (1,'Une île entre le ciel et l ''eau', \
:LOCALIMAGE)
IF SQL_ERROR% <> 0
 MESSAGE "INSERT IMAGE", SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
 F_CLOSE FILE%
 DISPOSE DATA%
 RETURN 1
ENDIF
F_CLOSE FILE%
DISPOSE DATA%

;SELECT
LOCAL DEST$(80),DATA%,SIZE%(4),NBREAD%(2),FILE%,NIL%,FNAME$, hbmp%
LOCAL SQL_IMAGE LOCALIMAGE

SQL_EXEC NS_FUNCTION IMAGEON
IF SQL_ERROR% <> 0
 MESSAGE "IMAGEON", SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
 RETURN 1
ENDIF
```

```
LOCALIMAGE.realsize = 30000
NEW LOCALIMAGE.realsize,LOCALIMAGE.PTR%
SQL_EXEC SELECT IMAGE INTO :LOCALIMAGE FROM T_IMAGE WHERE NUMERO = 1
IF SQL_ERROR% <> 0
 MESSAGE "SELECT IMAGE",SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
ELSE

FNAME$="(NS-BMP)\SOUVENIR.BMP"
FILE%=F_CREATE%(1,FNAME$)
INSERT AT END "FILE% "& FILE% TO LISTBOX1
F_BLOCKWRITE FILE%, LOCALIMAGE.PTR%, LOCALIMAGE.REALSIZE, NBREAD%
IF F_ERROR%
  MESSAGE"ERROR", "Failed to write " & FNAME$ &"!"
  F_CLOSE FILE%
  DISPOSE LOCALIMAGE.PTR%
  RETURN 1
ENDIF
HBMP%=CREATEBMP%(FNAME$)
BMPF = HBMP%
F_CLOSE FILE%
DISPOSE LOCALIMAGE.PTR%
ENDIF
DISPOSE LOCALIMAGE.PTR%
; ---- default mode
SQL_EXEC NS_FUNCTION IMAGEOFF
```

See also NSDBMS.NCL, SQL_ERROR%, SQL_ERRMSG$, TYPE_SQL_INSERT_BLOB%, TYPE_SQL_SELECT_BLOB%

NS_FUNCTION QUOTEOFF, QUOTEON

The QUOTEON mode automatically manages quotation marks in input parameters for string-type values. This function is used for replacing strings of characters at application runtime.

The QUOTEOFF mode disables this function.

Syntax **NS_FUNCTION QUOTEOFF**

and

NS_FUNCTION QUOTEON

Notes

1. QUOTEON is the mode by default.
2. Quotation marks must be entered by the user in QUOTEOFF mode and be handled automatically in QUOTEON mode.

Example

```
; This example allows to use host variables to create a SQL request

LOCAL A$,B$,C$,D$

MOVE "COL1" TO A$
MOVE "COL2" TO B$
MOVE "TABLE" TO C$
MOVE '"'&&"HELLO"&&'"' TO D$

SQL_EXEC NS_FUNCTION QUOTEOFF
IF SQL_ERROR% <> 0
   MESSAGE "ERROR QUOTEOFF",\
           SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
ENDIF
SQL_EXEC SELECT :A$,:B$ FROM :C$ WHERE :A$=:D$
;Equal to SELECT COL1, COL2
;FROM TABLE WHERE COL1 = ‘HELLO'
......
SQL_EXEC NS_FUNCTION QUOTEON
IF SQL_ERROR% <> 0
   MESSAGE "ERROR QUOTEON",\
           SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
ENDIF
```

See also SQL_ERROR%, SQL_ERRMSG$

NS_FUNCTION ROWCOUNT

Returns the number of rows affected by a query (SELECT, INSERT, UPDATE, etc.) or the number of the FETCH realized after a SELECT.

Syntax **NS_FUNCTION ROWCOUNT INTO** *:nbr-rows*

Parameter	*nbr-rows*	INT(4)	I	number of rows

Example 1

```
LOCAL ROWCOUNT%

SQL_EXEC DELETE FROM TABPRODUIT WHERE NOPROD >= 30 AND NOPROD < 40

SQL_EXEC NS_FUNCTION ROWCOUNT INTO :ROWCOUNT%
; If 10 records correpond to this filter, then 10 records will be delete, thus
; ROWCOUNT% will contain 10.
; If any record correspond to a filter thus ROWCOUNT% is equal to 0.
```

Example 2

```
local var1%
local  test$
LOCAL ROWCOUNT%
SQL_EXEC SELECT NUM, COL1 FROM BASE
IF SQL_ERROR% <> 0
    MESSAGE "Erreur ",SQL_ERRMSG$(SQL_ERROR%)
ENDIF


WHILE  SQL_ERROR% = 0
 SQL_EXEC FETCH INTO:var1%,:test$
 IF SQL_ERROR% <> 0
  BREAK
 ENDIF
 INSERT AT END "Var1"&&var1%&& "test"&&test$ TO LISTBOX1
ENDWHILE
SQL_EXEC NS_FUNCTION ROWCOUNT INTO :ROWCOUNT%
Message "Number of occurences = ", ROWCOUNT%
```

See also NS_FUNCTION ANSIOFF, NS_FUNCTION ANSION, SQL_ERROR%, SQL_ERRMSG$, COMPUTE

NS_FUNCTION RPCRETCODEOFF, RPCRETCODEON

The RPCRETCODEOFF mode lets you retrieve a return code during the execution of a stored procedure which does not return a code.

The RPCRETCODEON mode lets you specify the return code of SQL_RETURN directly in a stored procedure.

Syntax **NS_FUNCTION RPCRETCODEOFF**

and

NS_FUNCTION RPCRETCODEON

Notes

1. RPCRETCODEOFF is the default mode. Using SQL_RETURN in this mode returns a 0.
2. RPCRETCODEON can only be used for FUNCTIONS and not for INSTRUCTIONS.
3. See SQL_PROC, for more information about using this function.

Example

```
; See the SQL_PROC example
SQL_EXEC NS_FUNCTION RPCRETCODEOFF
SQL_EXEC SQL_PROC ...
...
SQL_EXEC NS_FUNCTION RPCRETCODEON
SQL_EXEC SQL_PROC ....
```

See also SQL_PROC, SQL_RETURN

NS_FUNCTION SETCURSORMODE

Sets the cursor mode used.

Syntax **NS_FUNCTION SETCURSORMODE** *:mode*

Parameter	*mode*	INT(4)	I	cursor mode used

Notes

1. There are two cursor modes. Each has a size of 1entry be default and their value is a constant set in NSDBMS.NCL.
2. For NSnnDB21 :

   - Mode by default (substitution) :
     `CONST DB_DB2_CURSOR_DEFAULT 3`

   - Mode binding :
     `CONST DB_DB2_CURSOR_NONE    0`

   Default mode : IN parameters are converted into character strings and replace the colon ":" in the SQL statement.

   Binding mode allows you to pass the values of IN parameters by address, like ? for a character string defined in the library.
3. You must be in **binding** mode if you want to use commands like RECORD, REEXECUTE or manipulate segments or images.

Example

```
LOCAL MODE%

; -----------------
;  Default mode
; -----------------
MOVE 'MORRIS' TO A$
SELECT * FROM TABLE WHERE COL = :A$
; => becomes
;    SELECT * FROM TABLE WHERE COL = 'MORRIS'

; ------------------
;  binding mode
; ------------------
MOVE DB_ODBC_CURSOR_BINDING% TO MODE%
SQL_EXEC NS_FUNCTION SETCURSORMODE : MODE%
SELECT * FROM TABLE WHERE COL = :A$
; => becomes
;    SELECT * FROM TABLE WHERE COL = ?
; the searches are done in binding mode

; --------------------------
;  return to default mode
; ---------------------------
MOVE DB_ODBC_CURSOR_DEFAULT% TO MODE%
SQL_EXEC NS_FUNCTION SETCURSORMODE : MODE%
```

See also RECORD, REEXECUTE , TYPE_SQL_INSERT_BLOB%, TYPE_SQL_SELECT_BLOB%, NSDBMS.NCL

NS_FUNCTION STATEMENT

Retrieves the full statement used in the query sent to the SQL engine. The SELECT is traced without the INTO clause, even if it is precised.

Syntax	**NS_FUNCTION STATEMENT INTO** : *query-string*			
Parameter	*query-string*	CSTRING	I/O	statement used in the query sent to the SQL engine

Note

1. ⚠ To trace the host variables correctly, use absolutely the **DB_DB2_CURSOR_NONE** mode.

Example 1

```
LOCAL VALUES$, PHRASE$

MOVE "HELLO" TO VALUES$
SQL_EXEC SELECT COL1 FROM TABLE WHERE COL2=:VALUES$

SQL_EXEC NS_FUNCTION STATEMENT INTO :PHRASE$
MESSAGE "The exact command is :",  PHRASE$

;PHRASE$ = SELECT COL1 FROM TABLE WHERE COL2='HELLO'
```

Example 2

```
LOCAL FATHER$, ID%, SON$, IND1%, IND2%, ID%, A$
LOCAL CSTRING Req$(2000)
LOCAL CURSORMODE%

CURSORMODE% = DB_DB2_CURSOR_NONE ;3
SQL_EXEC NS_FUNCTION SETCURSORMODE :CURSORMODE%

SQL_EXEC NS_FUNCTION QUOTEOFF

A$ ='"'&'PIERRE'&'"'
SQL_EXEC SELECT ID, FATHER, SON INTO :ID%, :FATHER$:IND1%, :SON$:IND2% FROM TOTO
\ WHERE FATHER =:A$
IF SQL_ERROR% <> 0
    MESSAGE "Error ",SQL_ERRMSG$(SQL_ERROR%)
ENDIF

SQL_EXEC NS_FUNCTION STATEMENT INTO :Req$
INSERT AT END Req$ TO LISTBOX1
; this way we can trace the value of A$ ("PIERRE" here)
; SELECT ID, FATHER, SON FROM TOTO WHERE FATHER ="PIERRE"
```

```
WHILE SQL_ERROR% = 0
 INSERT AT END FATHER$ TO LISTBOX1
 SQL_EXEC FETCH
ENDWHILE

SQL_EXEC NS_FUNCTION QUOTEON
CURSORMODE% = DB_DB2_CURSORDEFAULT ;0
SQL_EXEC NS_FUNCTION SETCURSORMODE :CURSORMODE%
A$ ='PIERRE'
SQL_EXEC SELECT ID, FATHER, SON INTO :ID%, :FATHER$:IND1%, :SON$:IND2% FROM TOTO
\ WHERE FATHER =:A$
IF SQL_ERROR% <> 0
    MESSAGE "Erreur ",SQL_ERRMSG$(SQL_ERROR%)
ENDIF
; All the variables have to be initialized even if their value will be null
SQL_EXEC NS_FUNCTION STATEMENT INTO :Req$
Insert AT END Req$ TO LISTBOX1
; SELECT ID, FATHER, SON FROM TOTO WHERE FATHER =@Param5
WHILE SQL_ERROR% = 0
 INSERT AT END FATHER$ TO LISTBOX1
 SQL_EXEC FETCH
ENDWHILE
```

NS_FUNCTION SETSTMTOPTION

Lets you specify the behavior of a cursor.

Syntax	**NS_FUNCTION SETSTMTOPTION** *:fOption% , :vParam%*		
Parameters	*:fOption%*	INT(4)	Option to change
	:vParam%	INT(4)	Value associated with option

Note

1. Use it with the following constants :

Option to change (*fOption%*)	**Value of the option (*vParam%*)**

Synchronous or asynchronous mode
`CONST DB_DB2_ASYNC_ENABLE   4`
	`CONST DB_DB2_ASYNC_ENABLE_OFF    0`
	`CONST DB_DB2_ASYNC_ENABLE_ON     1`

Bound column size for extented fetch `CONST DB_DB2_BIND_TYPE  5`	`CONST DB_DB2_BIND_BY_COLUMN      0`

Cursor concurrency
`CONST DB_DB2_CONCURRENCY  7`
	`CONST DB_DB2_CONCUR_READ_ONLY    1`
	`CONST DB_DB2_CONCUR_LOCK         2`
	`CONST DB_DB2_CONCUR_ROWVER       3`
	`CONST DB_DB2_CONCUR_VALUES       4`

Cursor type
`CONST DB_DB2_CURSOR_TYPE  6`
	`CONST DB_DB2_CURSOR_FORWARD_ONLY   0`
	`CONST DB_DB2_CURSOR_KEYSET_DRIVEN 1`
	`CONST DB_DB2_CURSOR_DYNAMIC       2`
	`CONST DB_DB2_CURSOR_STATIC        3`

Number of rows in keyset `CONST DB_DB2_KEYSET_SIZE    8`	`CONST DB_DB2_KEYSET_SIZE_DEFAULT  0`

Option to change (fOption%)	*Value of the option (vParam%)*
Maximum amount of data that the driver returns from a character or binary column `CONST DB_DB2_MAX_LENGTH     3`	`CONST DB_DB2_MAX_LENGTH_DEFAULT   0`
Maximum number of rows to return to the application for a SELECT statement `CONST DB_DB2_MAX_ROWS       1`	`CONST DB_DB2_MAX_ROWS_DEFAULT 0`

Scan SQL strings for escape clauses
`CONST DB_DB2_NOSCAN         2`
	`CONST DB_DB2_NOSCAN_OFF     0`
	`CONST DB_DB2_NOSCAN_ON      1`

Number of seconds to wait for an SQL statement `CONST DB_DB2_QUERY_TIMEOUT  0`	`CONST DB_DB2_QUERY_TIMEOUT_DEFAULT 0`

Does/does not retrieve data after it positions the cursor
`CONST DB_DB2_RETRIEVE_DATA  11`
	`CONST DB_DB2_RD_OFF    0`
	`CONST DB_DB2_RD_ON     1`

The number of rows in the rowset `CONST DB_DB2_ROWSET_SIZE   9`	`CONST DB_DB2_ROWSET_SIZE_DEFAULT 1`

To simulate positioned update and delete statement mode
`CONST DB_DB2_SIMULATE_CURSOR 10`
	`CONST DB_DB2_SC_NON_UNIQUE   0`
	`CONST DB_DB2_SC_TRY_UNIQUE   1`
	`CONST DB_DB2_SC_UNIQUE       2`

Use bookmarks with a cursor
`CONST DB_DB2_USE_BOOKMARKS   12`
	`CONST DB_DB2_UB_OFF          0`
	`CONST DB_DB2_UB_ON`

Example

```
LOCAL OPTION%, VALUE_OPTION%

OPTION% = DB_CURSOR_TYPE
VALUE_OPTION% = DB_CURSOR_FORWARD_ONLY

; the cursor should not be positioned in the rear
; this use a classical gestion of the internal cursor of the driver
SQL_EXEC NS_FUNCTION SETSTMTOPTION :OPTION%, :VALUE_OPTION%
```

See also NSDB2.NCL pour les constantes

NS_FUNCTION TRIMCHAROFF, TRIMCHARON

In TRIMCHARON mode , when a SELECT is executed, the blank spaces at the end of strings are removed. This is very useful when the array type is CSTRING or STRING, but it’s not applicable with CHAR or VARCHAR2.

In TRIMCHAROFF mode, the blank spaces are kept.

Syntax **NS_FUNCTION TRIMCHAROFF**

and

NS_FUNCTION TRIMCHARON

Note

1. TRIMCHAROFF is the mode by default.

Example

```
LOCAL C$, B$
; longstr is a varchar(2000)
;   &       TEST CHAR(10)
SQL_EXEC INSERT INTO TOTO (ID, TEST, LONGSTR) VALUES (1, 'A234567890', \
'lgstr23456789')
if sql_error% <> 0
  message 'error INSERT' , sql_errmsg$(sql_error%)
endif
SQL_EXEC INSERT INTO TOTO (ID, TEST, LONGSTR) VALUES (2, 'A2345', \
'lgstr2345    ')
if sql_error% <> 0
  message 'error INSERT' , sql_errmsg$(sql_error%)
endif
SQL_EXEC INSERT INTO TOTO (ID, TEST, LONGSTR) VALUES (3, 'A', 'lgstr        ')
if sql_error% <> 0
  message 'error INSERT' , sql_errmsg$(sql_error%)
endif
; Default mode
; ----- This loop will show <A234567890>
; <A2345              >
; <A                       >
; <lgstr23456789>,
;<lgstr234             >
;<lgstr                  >
SQL_EXEC SELECT TEST, LONGSTR FROM TOTO
if sql_error% <> 0
  message 'error SELECT' , sql_errmsg$(sql_error%)
endif
INSERT AT END ‘{J[C],B[LIGHTRED]}DEFAULT’ TO LISTBOX1
```

```
 WHILE SQL_ERROR% = 0
 SQL_EXEC FETCH INTO :C$, :B$
 if sql_error% <> 0
  BREAK
 endif
 INSERT AT END ‘{B[YELLOW]}CHAR=<’ & C$ & ‘>’&’’ TO LISTBOX1
 INSERT AT END ‘VARCHAR=<’ & B$ & ‘>’&’’ TO LISTBOX1
ENDWHILE
; ----- mode
SQL_EXEC NS_FUNCTION TRIMCHARON
if sql_error% <> 0
  message 'error TRIMCHARON' , sql_errmsg$(sql_error%)
endif
; -----  This will diplay  <A234567890>
; <A2345>
; <A>
; And <lgstr23456789>,
;<lgstr234>
;<lgstr>
INSERT AT END ‘{J[C],B[LIGHTRED]}TRIMCHARON’ TO LISTBOX1
SQL_EXEC SELECT TEST, LONGSTR FROM TOTO
if sql_error% <> 0
  message 'error SELECT' , sql_errmsg$(sql_error%)
endif
WHILE SQL_ERROR% = 0
   SQL_EXEC FETCH INTO :C$, :B$
 if sql_error% <> 0
  BREAK
 endif
 INSERT AT END ‘{B[YELLOW]}CHAR=<’ & C$ & ‘>’&’’ TO LISTBOX1
 INSERT AT END ‘VARCHAR=<’ & B$ & ‘>’&’’ TO LISTBOX1
ENDWHILE
; ---- Return to the default mode
SQL_EXEC NS_FUNCTION TRIMCHAROFF
if sql_error% <> 0
  message 'error TRIMCHAROFF' , sql_errmsg$(sql_error%)
endif
```

See also SQL_ERROR%, SQL_ERRMSG$