NatStar

Version 5.00 Edition 1

NS-DK

Version 5.00 Edition 1

NatWeb

Version 4.00 Edition 1

SQL Server

Ref. n° NS500SQLU00033

Contents

About this Manual

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

Supported configurations

Development environment

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

Client environment

Operating system	**DBMS drivers available**
Windows 32 bits	MS SQLServer 7.0 MS SQLServer 2000

Server environment

Operating system	**DBMS drivers available**
Windows NT Windows 2000 32 bits	MS SQLServer 7.0 MS SQLServer 2000

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.

What’s new in this edition

In this edition, the structure of the older manual entitled « Database Access Reference » has been modified to ease the using and to provide faster ways of finding the information you need. Thus, each library is described in a specifical manual.

Organization of the manual

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

Chapter 1	**SQL Server interface** Describes the functions of the NSnnMSxx library associated with the SQL SERVER 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 OS/2- PM 2.x or DOS- Windows 32 bits

UNIX

Indicates specific information on using the software under Unix systems

Indicates specific information on using the software under Macintosh

Chapter 1 SQL Server Interface

This chapter presents the libraries which allow you to interface Nat System development tools to with client versions of Microsoft SQL Server.

⚠ Les versions x.00 des outils de développement Nat System supportent en environnement Windows 32 bits : Microsoft SQL Server 7.0 et 2000.

⚠ To simplify documentation documentation NSnnMS70, NSnnMS2k and NSnnMS65 will be grouped under the generic name of NSnnMSxx.

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 extensions in this library.

Contents

Introduction

NSnnMSxx libraries allow to interface your applications with SQL Server applications. These libraries use the DBLIB version of Microsoft SQL Server libraries.

Correspondence between drivers and the versions of Microsoft SQL Server

The name of the driver should be used in particular with the instruction THINGS_DB_INIT for NatStar and the instruction SQL_INIT for NatStar, NatWeb and NS-DK

The following table indicates the versions of SQL Server and the corresponding drivers.

SQL Server version	**Driver**
Microsoft SQL Server 7.0	NSnnMS70.dll
Microsoft SQL Server 2000	NSnnMS2k.dll

☞ nn stands for the number of the version of the interface you have installed:

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

⚠ To simplify documentation documentation NSnnMS70, NSnnMS2k and NSnnMS65 will be grouped under the generic name of NSnnMSxx.

Previous versions of the drivers no longer supported by Nat System in the Windows 32 bit environment are installed in the CONTRIB(S) folder: NSw2MS65.dll et NSw2MS6.dll.

Installation

Copy the file NSnnMSxx.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 Microsoft SQL Server vendor.

Implicit Output Data Conversions

The SQL libraries use conversion functions of Microsoft SQL Server.

👓 Refer to the User Manual of Microsoft SQL Server to see the allowing conversions.

Nevertheless, for certain data, use the following conversions:

Microsoft SQL Server	**NCL**
DECIMAL, NUMERIC	CSTRING (*), DYNSTR (**),CHAR, INT, NUM
CHAR, VARCHAR	CSTRING (*), DYNSTR (**),CHAR
BINARY, VARBINARY	BLOBS, SEGMENT
TEXT, IMAGE	BLOBS, SEGMENT

(*) it is strongly recommended to use this type of data.

(**) The reading maximum size of a DYNSTR string is 255 characters for the previous versions of SQL Server 2000.

blob

A **blob** (or Binary Large Object) corresponds to a field of a database able to contain images, video and sound for multimedia applications.

Functional categories in the NSnnMSxx library

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

Initializing and stopping application use of the DBMS

Opening and closing a database

Managing the current database

Choose the DBMS and the base for an SQL command

Executing a SQL command: SELECT, INSERT, UPDATE, CREATE TABLE …

Managing the cursor

Running a stored procedure

Stored procedure

A **stored procedure** is a bit of code, written in a owner database language, stored in base.

Managing blobs

Configuring DBMS behavior

SQL query advanced management

Handling errors

Managing date format

Managing character sets

Command

NSnnMSxx libraries reference

SQL_INIT instruction

Loads the driver needed to use Microsoft SQL Server for a target.

Syntax **SQL_INIT** *DLL_name*

Parameters *DLL_name* CSTRING I name of driver to load

Notes

1. This must be the first SQL_ instruction called by any application that wants to use Microsoft SQL Server with NCL: it is responsible for loading the library.
2. The *DBMS-name* parameter should contain the name of the DLL used to access Microsoft SQL Server database:
    - NSnnMS70.DLL

      Interface from client version of Microsoft SQL Server 7.0
    - NSnnMS2k.DLL
      Interface from client version of Microsoft SQL Server 2000
      ☞ nn stands for the version number of the interface that you have installed :w2 for Windows 32 bits, w4 for Windows 64 bits.

Example

```
SQL_INIT "NSW2S70"  ; Load the Windows DLL
                    ; version of Sybase 7.0
IF SQL_ERROR% <> 0
   MESSAGE "An error occurred while loading the DLL",\
           "Check to make sure the directory " &\
           "containing the DLL is in your path "
ENDIF
...
SQL_STOP
```

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

SQL_STOP instruction

Unloads the Microsoft SQL Server driver and closes all open databases and cursors.

Syntax SQL_STOP

Applications must end with the SQL_STOP or equivalent instruction.

Example

See example of 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 database

Notes

1. Microsoft SQL Server allows you to open several databases on a network as follows.
2. The *logical-DBname* parameter specifies the logical database name
3. The *connection-string* parameter specifies the command string used to connect to a local or remote database, as follows:

```
"USERID[/PASSWORD][@<CONNECTOR>]\
[#LOGINTIMEOUT]
[;HOST=hostname][;APP=appname]"
```

where

`[USERID]` name of the user account

`[PASSWORD]` password for the user account.

`[@<CONNECTOR>]` SQL_EXEC is used to specify which open database will receive the query. If it is not specified, the local database will be accessed.

`[#LOGINTIMEOUT]` specifies the connection TIMEOUT in seconds. If it is not specified, the TIMEOUT is set to 120 seconds

`[;HOST=hostname]` specifies the name of the machine. This name will be visible with SP_WHO.

The name of the applicative does not contain the following characters: space, !, #, @ or /. Do not use spaces in hostname, use the underscore '_' to act like a space

```
SQL_OPEN ""db"", ""SQLUSER/passwd@dbhost;HOST=hostname""
```

`[;APP=appname]` specifies the name of the application. This name will be visible with SP_WHO.

The name of the applicative does not contain the following characters: space, !, #, @ or /. Do not use spaces in appname, use the underscore '_' to act like a space

```
SQL_OPEN ""db"", ""SQLUSER/passwd@dbhost;APP=appname""
```

4. If the database that you want to open is not the user's default database, SQL_OPEN must be followed by SQL_EXEC USE, which should indicate the name of the database that you want to use.
5. Si la base à ouvrir (paramètre *nom-logique-base*) n’existe pas sur le serveur, SQL_OPEN ouvre à la place la base par défaut de l’utilisateur.
6. If the *logical-Dbname* is not filled, SQL_OPEN opens the user's default database.
7. From NatStar 2.61, it is possible to connect anonymously to a SQL-Server database. Just don’t fill the `USERID[/PASSWORD]`fields.

Connection with Windows authentication:

```
sql_init "NS02MS2K"
if sql_error% < 0
  message 'error' , sql_errmsg$(sql_error%)
 endif
sql_open 'support', "@pollux"
if sql_error% < 0
  message 'open'&&sql_error% , sql_errmsg$(sql_error%)
 endif
```

Example

```
; ---- Connect to the Mdb1 database
;       in the local database
SQL_OPEN "Mydb1", "USR1/PSWD"
IF SQL_ERROR% <> 0
   MESSAGE "Mydb1 error", SQL_ERRMSG$(SQL_ERROR%)
ENDIF

; ---- Connect to the Mdb2 database
;       in the SERV2 database

SQL_OPEN "Mydb2", "USR1/PSWD1@SERV2"
IF SQL_ERROR% <> 0
   MESSAGE "Mydb2 error", SQL_ERRMSG$(SQL_ERROR%)
ENDIF
...
SQL_CLOSE "Mydb2"
SQL_CLOSE "Mydb1"
```

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

SQL_CLOSE instruction

Closes a connection of the database.

Syntax	**SQL_CLOSE** *logical-DB-name*		
Parameter	*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 example of SQL_OPEN instruction.

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

AT command

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 "MYDB1", "SCOTT/TIGER"         ; Local database
SQL_OPEN "MYDB2", "SCOTT/TIGER"@MYSERV  ; Remote database
SQL_OPEN "MYDB3", "SCOTT/TIGER"@MYSERV  ; Remote database

SQL_EXEC SELECT....                      ; SELECT on MYDB3

SQL_EXEC AT MYDB2 SELECT....             ; SELECT on MYDB2
SQL_EXEC AT MYDB2 SELECT....             ; SELECT on MYDB2

SQL_EXEC FETCH...                        ; FTCH MYDB3
```

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

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 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), 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 chapters 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       VARCHAR(10),\
                              AGE        NUMBER,      \
                              CHILDNAME  VARCHAR(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:PERE$:IND1%,:AGE%,:SON$: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 statement.: 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
cursor_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 functionalities 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% = -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 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 very long SQL statement : 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
```

6. This array of segments should have an **index that is greater than** the number of variables used (the last element contains 0).This is why we advise initially filling this array (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 handle-cursor
   SQL_EXEC FETCH ... USING handle-cursor
   ```
2. A cursor is an internal resource managed by the NSnnMSxx 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 NSnnMSxx 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%.

11. Only NSnnMS65 has cursors which use a single physical connection. The other drivers use a physical connection for each execution of SQL_OPENCURSOR%.

    Therefore, if one cursor is used for the INSERT and another is used for the SELECT, only NSnnMS65 can select the INSERT.

Example

See the example of 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 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 handle-cursor
   SQL_EXEC FETCH ... USING handle-cursor
   ```
2. A cursor is an internal resource managed by the NSnnMSxx 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

See the example of 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** *handle-cursor*

Parameters

handle-cursor	INT(4)	I	handle of the 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 can be used with all DBMS drivers.
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 NatWeb errors:

   0 "NSSQLE000 ** UNUSED NATSYS ERROR CODE"

   Cause: No error occurred, the execution was carried out correctly.

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

   Cause: There are no rows or the last row was reached after a FETCH or SELECT.

   +200 "NSSQLW200 ** COMPUTE RESULT IN PROGRESS"

   Cause: The SELECT command containing the COMPUTE is being executed.

   -32001 "NSSQLE001 ** HEAP ALLOCATION ERROR"

   Cause: Internal memory allocation error.

   -32002 "NSSQLE002 ** DYNAMIC ALLOCATION ERROR"

   Cause: Internal memory allocation error.

   -32003 "NSSQLE003 ** DYNAMIC FREE STORAGE ERROR "

Cause: Internal memory allocation error.

-32004 "NSSQLE004 ** NO MORE CURSORS AVAILABLE"

Cause: Too many cursors opened simultaneously.

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

Cause: Attempt to free a cursor when there are none.

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

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

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

Cause: Too many variables in the INTO clause.

-32008 "NSSQLE008 ** MISSING HOST VARIABLE AFTER"

Cause : Erreur de syntaxe dans une clause INTO. Une virgule de continuation est trouvée sans variable derrière.

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

Cause: Syntax error in the INTO clause. A serial comma exists without a variable after.

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

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

Cause: Because a new SQL command was received by the cursor, the system closed the cursor containing the 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 the UPDATE WHERE CURRENT OF command.

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

Cause : The FETCH as well as the preceding SELECT do not have any output variables declared (INTO clause).

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

Cause : A FETCH without a preceding SELECT, or a cursor closed by the system between a SELECT and FETCH command.

-32016 "NSSQLE016 ** INVALID SQL DATA TYPE "

Cause : Invalid output data type.

-32017 "NSSQLE017 ** INVALID DATA CONVERSION REQUESTED"

Cause : Invalid output conversion.

STRING -> NUM

NUM -> STRING

REAL -> INTEGER

INTEGER -> REAL

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

Cause : Invalid length for the given data type (ex: real 48 characters).

-32019 "NSSQLE019 ** INVALID DECIMAL PACKED FORMAT"

Cause : The conversion of data to decimal format could not be done.

+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"

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

Cause : The executed instruction has no effects.

-32023 "NSSQLE023 ** TOO MANY OPENED DATABASES"

Cause : Too many 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 MSSQLSERVER LOGIN"

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

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

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

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

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

-32031 "NSSQLE031 ** MSSQLSERVER BUFFER FILL ERROR"

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

-32032 "NSSQLE032 ** RPC PARAMETER NAME EXPECTED"

Cause : Remote procedure call : parameters 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'SIZE TOO SMALL TO HOLD VALUE"

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

-32058 "NSSQLE058 ** RPC : INIT (MSSQLSERVER) ERROR"

-32059 "NSSQLE059 ** RPC : PARAMETERS FOUND BUT NO VALUE 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 ** SYBEXEC : INCOMPATIBLE CURSOR MODE"

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

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

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

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

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

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

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

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

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

-32079 "NSSQLE079 ** SYBEXEC : 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 USER LENGTH"

-32093 "NSSQLE093 ** INVALID PASSWORD LENGTH"

-32094 "NSSQLE094 ** INVALID SERVER LENGTH"

-32096 "NSSQLE096 ** INVALID LOGINTIME LENGTH"

-32097 "NSSQLE097 ** LOGINTIME MUST BE DIGIT NUMBER"

-32098 "NSSQLE098 ** DATABASE ALREADY OPENED"

-32099 "NSSQLE099 ** INVALID LENGTH FOR TDS PACKET SIZE"

-32101 "NSSQLE0101 ** TDS PACKET SIZE TIME MUST BE DIGIT VALUE"

-32102 "NSSQLE0102 ** UNABLE TO OPEN FILE"

-32103 "NSSQLE0103 ** NO MEMORY AVAILABLE"

-32104 "NSSQLE0104 ** NO CONNECTION AVAILABLE TO UPDATE IMAGE/TEXT"

-32105 "NSSQLE0105 ** UNABLE TO COMMIT THE TRANSACTION (TM MODE)"

-32106 "NSSQLE0106 ** UNABLE TO ROLLBACK THE TRANSACTION (TM MODE)"

-32107 "NSSQLE0107 ** UNABLE TO RELEASE THE TRANSACTION (TM MODE)"

-32108 "NSSQLE0108 ** UNABLE TO BEGIN THE TRANSACTION (TM MODE)"

-32109 "NSSQLE0109 ** UNABLE TO ENLIST THE TRANSACTION (TM MODE)"

-32110 "NSSQLE0110 ** UNABLE TO ENLIST THE TRANSACTION (XA MODE)"

-32111 "NSSQLE0111 ** INVALID CHARACTER SET IDENTIFIER"

-32112 "NSSQLE0112 ** INVALID DATE FORMAT"

-32113 "NSSQLE0113 ** INVALID TIME FORMAT, USING DEFAULT FORMAT: HMSL"

-32114 "NSSQLE0114 ** INVALID HOSTNAME LENGTH, TOO LONG"

-32115 "NSSQLE0115 ** INVALID APPLICATION NAME, TOO LONG"

-32116 "NSSQLE0116 ** ERROR PARSING HOSTNAME OR APPLICATION STRING"

5. With Microsoft SQL Server, the notion of errors and warnings are completly different from the previous versions. Now, most of the older warnings are errors.

Example

```
...
SQL_EXEC SELECT COL1 FROM TAB1
IF SQL_ERROR% < 0
      MESSAGE "Error  : " & SQL_ERROR%,\
            SQL_ERRMSG$(SQL_ERROR%)
ELSEIF SQL_ERROR% > 0
      MESSAGE "Warning : " & SQL_ERROR%,\
            SQL_ERRMSG$(SQL_ERROR%)
ELSE
      MESSAGE "OK", "No error"
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 NSnnMSxx DLL when the error occurred.
2. See SQL_ERROR% for a detailed list of error codes and messages.

Example

See the example of SQL_ERROR% function.

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*

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

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

Parameters *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. The SQL_PROC command accepts null indicators. The null indicator is an output variable (OUT) which may be initialized with any variable or not be initialized at all. It is the value returned that is significant. If the value returned is 1, this means that the associated parameter is not null, if 0 it means that it is nul.
5. **Important** :
   - 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 whit output parameters (OUT), you must use SQL_RETURN to retrieve any errors.

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

Example

```
; Creation of the procedure

SQL_EXEC CREATE PROCEDURE RPCTEST(@PARAM1 INT OUT, \
                          @PARAM2 INT OUT, \
                          @PARAM3 INT OUT, \
                          @PARAM4 INT OUT) \
 AS \
 BEGIN \
 SELECT @PARAM1 = @PARAM1 + @PARAM4 \
 SELECT @PARAM2 = @PARAM2 + @PARAM4 \
 SELECT @PARAM3 = @PARAM3 + @PARAM4  \
 SELECT AU_LNAME FROM AUTHORS  \
 RETURN 200 \
 END


IF SQL_ERROR% <> 0
    MESSAGE "Error create proc with FETCH",SQL_ERRMSG$(SQL_ERROR%)
      sql_exec ns_function statement into :s$
      message "",s$
ENDIF

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

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

SQL_EXEC SQL_PROC RPCTEST (@PARAM1 OUT, @PARAM2 OUT, \
                           @PARAM3 OUT, @PARAM4 OUT )     \
         VALUES ( :A% , :B% , :C% , :D% )

if SQL_ERROR% <> 0
   MESSAGE "Error SQL_PROC RPCTEST",SQL_ERRMSG$(SQL_ERROR%)
ENDIF

; Retrieve the result of the procedure in NCL variables
; Here we retrieve the "Message retrieved by a FETCH" in B$
; Retrieve the result of a SELECT executed on AUTHORS
WHILE SQL_ERROR% = 0
  SQL_EXEC FETCH INTO :B$
  IF SQL_ERROR% <> 0
      if SQL_ERROR% < 0
         MESSAGE "Error AUTHORS:B$",SQL_ERRMSG$(SQL_ERROR%)
      ENDIF
  ELSE
         INSERT AT END B$ TO LB1
  ENDIF
ENDWHILE

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

```
; Display the OUT parameters of the procedure
; A%, B%, C% have been incremented by 1 (by d%)
INSERT AT END "Result"&&A%&&B%&&C% TO LB1

; Display R% (= 200)
INSERT AT END  "Return code from RPCTEST"&&R% TO LB1
```

See also SQL_RETURN, SQL_SYPROC, SQL_ERROR%, SQL_ERRMSG$

SQL_RETURN command

Retrieves the error code from the last procedure executed on a Microsoft SQL Server server.

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

Parameter *return-code* INT(4) O return code

Notes

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).
3. The syntax checker does not check anything typed after 'SQL_EXEC'. So, if SQL_RETURN is misspelled, a syntax error will not be generated but SQL errors will occur.
4. When processing a stored procedure, all of the SQL_EXEC commands must be verified by SQL_ERROR% to ensure the correct execution of the procedure.

Example

```
; Retrieve the error code from RPCTEST (value 0)
LOCAL R%
SQL_EXEC SQL_RETURN:R%

IF SQL_ERROR% <> 0
      ; Error handling
ENDIF

; Display R%
MESSAGE "Return code from RPCTEST",R%
```

See also SQL_PROC, SQL_SYPROC, SQL_ERROR%, SQL_ERRMSG$

SQL_SYPROC command

Note

☞ This command is the same as SQL_PROC.

See also SQL_PROC, SQL_RETURN

TYPE_SQL_INSERT_BLOB%, TYPE_SQL_SELECT_BLOB% types for blobs

Enables management of binary large objects, insertion with TYPE_SQL_INSERT_BLOB% and selection with TYPE_SQL_SELECT_BLOB%.

Notes

1. Two 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. The images inserted with TYPE_SQL_INSERT_BLOB% are not limited in size.
3. The selection of images with TYPE_SQL_SELECT_BLOB% is limited by default to 1 Mo. However, this size should be modified by NS_FUNCTION CHANGEOPTION TEXTSIZE instruction.
4. SQL-Server 2000 can't read more than 255 characters from a VARCHAR column. To remedy this :
   - create a column of TEXT type
   - Use TYPE_SQL_CSTRING% to insert data
   - Use TYPE_SQL_SEGMENT% to select data

   To insert/select TEXT data larger than 4 K, use NS_FUNCTION CHANGEOPTION to enlarge TEXTSIZE and TEXTLIMIT options.
5. Some specific using rules for SQL_Server 2000 :
   - The table must contain only one column of IMAGE or TEXT type.
   - The SELECT clause must contain only one column : IMAGE column.
   - We have to force IMAGE column to NULL by an UPDATE clause before the insertion.
   - The insertion ismade by a SELECT clause (and not with an INSERT clause) with *Type_Var* TYPE_SQL_INSERT_BLOB%.

Example 1

```
LOCAL NCLVAR HL[4]
LOCAL INT IMAGNO
LOCAL DESCRIP$
LOCAL FIMAGE$
LOCAL INT J
LOCAL SQL$
LOCAL BMP%
LOCAL CURSORMODE%
LOCAL Opt$
LOCAL Val%

; to able to insert blobs we have to be in CURSORMODE% = 0 DB_MS_CURSORDEFAULT
CURSORMODE% = DB_MS_CURSORDEFAULT; 0
SQL_EXEC NS_FUNCTION SETCURSORMODE :CURSORMODE%

;;;;; To select binaries >  1 Mega, increase text size !!!
;Opt$ = 'TEXTSIZE'
;Val% = 3000000
;SQL_EXEC NS_FUNCTION CHANGEOPTION :Opt$, :Val%
;if (sql_error% < 0)
; message 'textsize', sql_errmsg$ (sql_error%)
;endif

SQL_EXEC DROP TABLE BIGIMAGE
; Creation of the table
SQL_EXEC CREATE TABLE BIGIMAGE(NUMERO INTEGER,\
DESCRIPTION VARCHAR(255), \
COLIMAGE IMAGE NULL)

; ---- Insert of TINTIN.BMP in table BIGIMAGE
SQL_EXEC INSERT INTO BIGIMAGE \
VALUES(1,'This is a big picture > 32000 bytes', NULL)
SQL_EXEC UPDATE BIGIMAGE SET COLIMAGE = NULL WHERE NUMERO = 1

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$="SELECT COLIMAGE INTO :FROM BIGIMAGE WHERE NUMERO = 1"
SQL_EXEC_LONGSTR @SQL$, @HL, DEFAULT_CURSOR
if (sql_error% < 0)
 message 'Select error', sql_errmsg$ (sql_error%)
endif
; ---- 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$="SELECT COLIMAGE INTO : FROM BIGIMAGE"
SQL_EXEC_LONGSTR @SQL$, @HL, DEFAULT_CURSOR

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

Example 2

Creation and insertion

```
LOCAL NCLVAR VARLIST[3]
LOCAL SQL_str$
LOCAL  var$ (30)
LOCAL CHAINE2K LgIN
LOCAL var1%

sql_exec drop table aelta
if sql_error% <> 0
  message 'error drop' , sql_errmsg$(sql_error%)
endif
sql_exec create table aelta (ID INT, LONGSTR  TEXT  NULL)
if sql_error% <> 0
  message 'error create' , sql_errmsg$(sql_error%)
endif

FILL @LgIN, SIZEOF CHAINE2K, 0
INSERT AT END "inout string LGIN" TO LISTBOX1
LGIN.str$="Insertion of long strings"
LGIN.str$(45..90) ="which is not truncated after the binary 0"
LGIN.str$(250..350) ="this is a string after the 250 charcaters bypassing the
255 limits"
INSERT AT END LGIN.str$(0..249) TO LISTBOX1
INSERT AT END LGIN.str$(250 ..350) TO LISTBOX1

Fill @VARLIST, SIZEOF VARLIST, 0
var1%=2

sql_str$ = "INSERT INTO AELTA (ID, LONGSTR) VALUES (:, : )"

VARLIST[0].PTR_VAR= @var1%
VARLIST[0].TYPE_VAR= TYPE_SQL_INT%
VARLIST[0].SIZE_VAR= SIZEOF var1%

VARLIST[1].PTR_VAR= @LgIN
VARLIST[1].TYPE_VAR= TYPE_SQL_CSTRING%
VARLIST[1].SIZE_VAR= SIZEOF CHAINE2K


SQL_EXEC_LONGSTR @SQL_str$, @varlist, -1
IF SQL_ERROR% <> 0
    MESSAGE "Erreur ",SQL_ERRMSG$(SQL_ERROR%)
 ENDIF
```

Selection

```
LOCAL NCLVAR VARLIST[3]
LOCAL SQL$
LOCAL CHAINE2K LgOut

Fill @VARLIST, SIZEOF VARLIST, 0
VARLIST[0].PTR_VAR = @LgOut
VARLIST[0].TYPE_VAR = TYPE_SQL_SEGMENT%
VARLIST[0].SIZE_VAR = SIZEOF CHAINE2K
SQL$="SELECT LONGSTR INTO : FROM AELTA WHERE ID = 2"
SQL_EXEC_LONGSTR @SQL$, @VARLIST, DEFAULT_CURSOR
if sql_error% <> 0
  message 'error' , sql_errmsg$(sql_error%)
endif

INSERT AT END "output string LGOUT" TO LISTBOX1
INSERT AT END LgOut.str$(0..249) TO LISTBOX1
INSERT AT END LgOut.str$(250 ..350) TO LISTBOX1
```

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

COMPUTE command

Retrieves the number of occurrences. The COMPUTE command in Microsoft SQL Server must be preceded by a SQL_EXEC instruction and respect the following syntax.

Syntax **COMPUTE** *Aggregate-Function* (**Column-Name**)

Notes

1. The SELECT command must be made of a incremental function (COUNT, MIN, MAX, AVG, SUM).
2. The execution of a search command containing COMPUTE will cause an error: NSSQLW200 ** COMPUTE RESULT IN PROGRESS.
3. The number of records retrieved by COMPUTE and ROWCOUNT may differ. ROWCOUNT may retrieve the number of FETCHs carried out, while COMPUTE retrieves the number of occurrences.
4. Cursor mode may be set to 0.
5. ☞ For more informations, see NS_FUNCTION SETCURSORMODE.

Example

```
LOCAL ROWS_AFFECTED%, A$, B$, cursormode%
cursormode% = 0
sql_exec NS_FUNCTION SETCURSORMODE :cursormode%

SQL_EXEC SELECT AU_ID, AU_LNAME INTO :A$,:B$,:ROWS_AFFECTED% FROM AUTHORS \
COMPUTE COUNT(AU_ID)
if sql_error% <> 0
  message 'error' , sql_errmsg$(sql_error%)&&sql_error%
endif

WHILE SQL_ERROR%=0 OR SQL_ERROR%=200
      IF SQL_ERROR% =200
             MESSAGE "Number of lines : ", ROWS_AFFECTED%
      ELSE
             INSERT AT END  "ID et NAME"&&A$ && B$ TO LISTBOX1
      ENDIF
      SQL_EXEC FETCH
ENDWHILE
```

See also NS_FUNCTION ROWCOUNT, NS_FUNCTION SETCURSORMODE

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. **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
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 NSnnMSxx, the commands are executed dynamically and use are based on DBLIB. Some SQL commands are not accessible by in *Transact SQL*, which is the case of the CURRENT OF clause
2. When using the NSnnMSxx, be sure to respect the following rules:

   - You must be in a transaction before you can use **FOR UPDATE** or **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%
LOCAL CURSORMODE%

CURSORMODE% = 2
; CURSORMODE% = 0 (for MSNNMS65)
; Change cursor mode
SQL_EXEC NS_FUNCTION SETCURSORMODE:CURSORMODE%
...
SQL_EXEC BEGIN TRANSACTION
; ---- Select and lock the columns
SQL_EXEC SELECT SENIORITY , BONUS
         INTO:SEN%      \
         FROM PERSONNEL             \
         WHERE SENIORITY >= 2 AND SENIORITY < 25 \
         FOR UPDATE

WHILE SQL_ERROR% = 0
  EVALUATE SEN%
    WHERE 2 TO 4, 6 TO 9
      ; ---- update for 2 to 4 and 6 to 9 years
      SQL_EXEC UPDATE PERSONNEL SET BONUS = BONUS * 1.02 CURRENT OF
ENDWHERE
    WHERE 5,10,15,20 TO
      ; ---- update for 5, 10,15 and 20 years
      SQL_EXEC UPDATE PERSONNEL SET BONUS = BONUS *1.03 CURRENT OF
ENDWHERE
    ELSE
      ; ---- update other years
      SQL_EXEC UPDATE PERSONNEL SET BONUS = BONUS * 1.01 CURRENT OF
ENDEVALUATE
  ; ---- read next record
  SQL_EXEC FETCH
ENDWHILE

; ---- to unlock and commit changes
SQL_EXEC COMMIT TRAN

; Change cursor mode (return to default)
CURSORMODE% = 0
; CURSORMODE% = 2 (for MSnnMS65)
SQL_EXEC NS_FUNCTION SETCURSORMODE:CURSORMODE%
...
...
```

See also

See Microsoft SQL Server documentation for more information about locking and transactions.

RELATIVE command

Positions the pointer on a specific record. This command is only available in **Buffering mode** and **Scrolling mode**.

Syntax

RELATIVE TO *:pos-fetch%*

INTO *[:Host_Variable] [:Host_Variable,...] [:Indicator]*

Parameter				
	pos-fetch	INT(4)	I/O	position of the record in the selection

Notes

1. In **Buffering mode**, the pointer can only be positioned inside the current buffer. If the requested record is not in the current buffer, the error message is returned:
   ```
       : -77 SYBEXEC : INVALID ROW NUMBER
   ```
2. In **Scrolling mode**, the pointer can be positioned on any record in the table at any time.
   - The host variable specified after the RELATIVE TO command specifies the position of the record in the selection.
   - The host variables specified after INTO will be used to retrieve the record's values (see FETCH).

Example

```
LOCAL name$
local cursormode%, posSelect%
; scrolling mode
cursormode% = 2
sql_exec NS_FUNCTION SETCURSORMODE :cursormode%
; ---- Selection with a lock on SELECT columns
SQL_EXEC SELECT AU_lname INTO :name$ FROM AUTHORS WHERE state = 'CA'
if sql_error% <> 0
  message 'error' , sql_errmsg$(sql_error%)
endif
INSERT AT END  "NAME ="&& name$ TO LISTBOX1
posSelect% = 10
SQL_EXEC RELATIVE TO :posSelect% INTO :name$
if sql_error% <> 0
 message 'error' , sql_errmsg$(sql_error%)
endif
INSERT AT END  "NAME ="&& name$ TO LISTBOX1
; result : White Yokomoto
```

See also NS_FUNCTION SETCURSORMODE, NS_FUNCTION SETBUFFERSIZE

NS_FUNCTION extensions

The NS_FUNCTION have been developed to increase the functionalities of databases's interface.

Syntax SQL_EXEC **NS_FUNCTION** *command*

Note

1. *command* is one of the commands described further on.

See also SQL_ERROR%, SQL_ERRMSG$

NS_FUNCTION ANSIOFF, ANSION

☞ The two NS_FUNCTION ANSIOFF and ANSION have been created to bypass SQL_EXEC UPDATE … WHERE … setting 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 "Aucun enregistrement mis à jour",
           SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
ENDIF

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

See also SQL_ERROR%, SQL_ERRMSG$

NS_FUNCTION ASYNCOFF, ASYNCON

The NS_FUNCTION ASYNCOFF allows asynchronous mode to be set to OFF (default value). In synchronous mode the client application waits for a response indicating the end of processing from the server before continuing.
The NS_FUNCTION ASYNCON allows asynchronous mode to be set to ON. In asynchronous mode the application may continue with other processing while waiting for the server to finish processing.

Syntax

NS_FUNCTION ASYNCOFF

and

NS_FUNCTION ASYNCON

Notes

1. In ASYNCHRONOUS, users do not have to wait for a query to finish executing: they will regain control as soon as the query is invoked. In SYNCHRONOUS mode, the application has to wait for a query to finish executing before it can regain control.
2. The default execution mode is SYNCHRONOUS
3. The INTO:host_variable clause can not be used in ASYNCHRONOUS mode. Executing a SELECT...INTO or FETCH...INTO command results in the following error:

```
NSSQLE081 DATA NOT READY TO RESULT PROCESSING
```

Example

```
SQL_EXEC NS_FUNCTION ASYNCOFF ; Sets the execution mode to SYNCHRONOUS
SQL_EXEC NS_FUNCTION ASYNCON  ; Sets the execution mode to ASYNCHRONOUS
```

See also

NS_FUNCTION KILLQUERY

NS_FUNCTION CALLBACK

This function enables the redirection of Microsoft SQL Server error messages to an application window. Each time an error message appears, Microsoft SQL Server sends an event to the 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.
2. 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.
3. 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.
4. To cancel this function, set the window handle to zero.
5. 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_MS_SERVER_STRUCT
  INT     msgnumber(4)
  INT     state(4)
  INT     severity(4)
  CHAR    text(1024)
  INT     textlen(4)
  CSTRING svrname(131)
  INT     svrnlen(4)
  CSTRING proc(131)
  INT     proclen(4)
  INT     line(4)
  INT     status(4)
  CSTRING sqlstate(7)
  INT     sqlstatelen(4)
ENDSEGMENT
```

```
SEGMENT DB_MS_CLIENT_STRUCT
 INT SEVERITY(4)
 INT MSGNUMBER(4)
 CHAR MSGSTRING(1024)
 INT MSGSTRINGLEN(4)
 INT OSNUMBER(4)
 CHAR OSSTRING(1024)
 INT OSSTRINGLEN(4)
ENDSEGMENT
```

The constants are :

```
CONST CLIENTMSG 1
CONST SERVERMSG 2
```

6. Two types of messages are sent.
7. **If the call NS_FUNCTION CALLBACK is carried out without USING**, the window manages error messages with the cursor. In this situation, it is the default cursor error messages that are redirected. It is useful, therefore, to use a cursor window.
8. ☞ 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
; cancel of the redirection
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 CATCHERR window
; ---------------------------------------------
LOCAL MESSAGETYPE%(4)
LOCAL PTR%(4)
MOVE PARAM12% TO MESSAGETYPE%
MOVE PARAM34% TO PTR%

 IF MESSAGETYPE% = CLIENTMSG
   INSERT AT END "ERROR : " &&DB_DB2_CLIENT_STRUCT(PTR%).nativeCode TO SELF%
   INSERT AT END "sqlstate : " &&DB_DB2_CLIENT_STRUCT(PTR%).sqlstate TO SELF%
   INSERT AT END "MSGSTRING " & DB_DB2_CLIENT_STRUCT(PTR%).MSGSTRING TO self%
 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

Enables switching from one database to another among the application's open databases.

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

Syntax **NS_FUNCTION CHANGEDBCNTX** *: logical-DBname*

Parameter	*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"

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

; the current database becomes pubs
SQL_EXEC NS_FUNCTION CHANGEDBCNTX :DATABASENAME$
```

See also SQL_OPEN, SQL_CLOSE, NS_FUNCTION GETDBNAME, AT

NS_FUNCTION CHANGEOPTION

Changes the connection parameters between the server and the client.

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

Parameters

parameter	CSTRING	I	parameter to change
option	*	I	value assigned to the parameter

*: depending on the value of *parameter*, the second variable will either be an INTEGER or a STRING (cf. table below).

Notes

1. The ***parameter*** variable can take the following values :

LOGINTIME Timeout delay for login (in seconds). By default the delay is 60 seconds.

TIMEOUT Timeout delay for any SQL operation (in seconds). By default the delay is 120 seconds. Sends a message at the end of a period of time specified by the *option* variable if execution of an SQL command does not run correctly. This parameterization may be useful during searches on large databases.

TEXTSIZE Maximum size of images/text to be extracted (in octets). By default, the size is limited to 1 Mb.

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

value% =  2147483647 ; max value !
option$ = "TEXTSIZE"
SQL_EXEC NS_FUNCTION CHANGEOPTION :option$, :value%
if (sql_error% ...)
```

TEXTLIMIT Maximum size of images/text to be inserted. By default, updates are carried out via SQL Server.

MULTICURSORS Simultaneous opening of several cursors: 1 (TRUE) or 0 (FALSE). By default, the value of MULTICURSORS is zero.

2. ⚠ To use several cursors simultaneously with SQL Server it is vital to use the NS_FUNCTION CHANGEOPTION with the MULTICURSORS parameter.

Example 1

```
LOCAL A$
LOCAL TMPS%

MOVE "TIMEOUT" TO A$
MOVE 30 TO TMPS%
SQL_EXEC NS_FUNCTION CHANGEOPTION :A$, :TMPS%
```

Example 2

```
LOCAL CURSOR1%,CURSOR2%, MODE%
LOCAL option$, value%
LOCAL INT ID, INT  ID2
;Call the ns_function ChangeOption with the option 'MultiCursors'
;if you have a cursor in cursormode DEFAULTCURSORMODE (0)
option$ = 'MULTICURSORS'
value% = 1
sql_exec ns_function changeoption :option$, :value%
if sql_error% <> 0
  message 'error changeoption' , sql_errmsg$(sql_error%)
endif

CURSOR1% = Sql_OpenCursor%
MODE%    = 0
Sql_Exec NS_FUNCTION SETCURSORMODE :MODE% USING CURSOR1%
if sql_error% <> 0
  message 'error SETCURSORMODE' , sql_errmsg$(sql_error%)
endif
Sql_Exec Select EMPNO from EMP USING CURSOR1%
if sql_error% <> 0
  message 'error Select AUTHORS' , sql_errmsg$(sql_error%)
endif
ID = 0
Sql_Exec Fetch Into :ID USING CURSOR1%
if sql_error% <> 0
  message 'error Fetch AUTHORS' , sql_errmsg$(sql_error%)
endif
Insert At End ID To LISTBOX1
CURSOR2% = Sql_OpenCursor%
Sql_Exec Select ID from AELTA USING CURSOR2%
if sql_error% <> 0
  message 'error Select AELTA' , sql_errmsg$(sql_error%)
endif
ID2 = 0
Sql_Exec Fetch Into :ID2 USING CURSOR2%
if sql_error% <> 0
  message 'error Fetch AELTA' , sql_errmsg$(sql_error%)
endif
Insert At End ID2  To LISTBOX1
Sql_Exec Fetch Into :ID USING CURSOR1%
if sql_error% <> 0
  message 'error create' , sql_errmsg$(sql_error%)
endif
Insert At End ID To LISTBOX1
```

NS_FUNCTION CHARTOHEXAOFF, CHARTOHEXAON

Sets CHARTOHEXAOFF mode, which disables automatic conversions of 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 default mode.
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. If the character string begins with 0x, then it is considered to contain a binary value.
3. Strains on character strings of STRING or CSTRING types :
   - The first two characters are: "0x" or "0X",
   - The following characters values: [0-9] or [A-F],
   - The number of character is even.
4. ☞ In CHARTOHEXAON mode, the insertion in a column of a 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 a SELECT * FROM TABLE WHERE col1 = 0x11FF
SQL_EXEC NS_FUNCTION Statement INTO :Req$
Insert AT END Req$ TO LISTBOX1
; result of NS_FUNCTION Statement SELECT * FROM TABLE WHERE col1 = 0x11FF

; return to the active mode by default
SQL_EXEC NS_FUNCTION CHARTOHEXAOFF

SELECT * FROM TABLE WHERE col1 = :A$

SQL_EXEC NS_FUNCTION Statement INTO :Req$
Insert AT END Req$ TO LISTBOX1
; result of NS_FUNCTION Statement SELECT * FROM TABLE WHERE col1 = '0x11FF'
;'0x11FF' is considered as a string not as an Hexadecimal value
```

See also SQL_ERROR%, SQL_ERRMSG$

NS_FUNCTION DATAREADY

When a query is executed in asynchronous mode, this function enables you to know whether or not the processing is finished.

Syntax	**NS_FUNCTION DATAREADY INTO** *:dataready*		
Parameter	*dataready*	NCL Type	Boolean flag 1 : finished 0 : processing
See also	NS_FUNCTION ASYNCOFF, NS_FUNCTION ASYNCON		

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*

Parameters	*nbr-errors*	INT(4)	O	number of errors or error messages encountered while executing a query

Notes

1. The error stack (and the other driver specific buffers) is refreshed each time a new query is called.

2. ☞ The error message « General error on SQL Server, look up SQL Server messages » corresponds to an OS error and not to a database manager error. This error is not fatal for transactions.

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 its number in ERROR%
 INSERT AT END "ERROR" && I% && SQL_ERRMSG$(ERROR%) TO LISTBOX1
 I% = I% + 1
ENDWHILE
ENDIF
```

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

NS_FUNCTION GETCURRENTDBCNTX

Returns the logical name of the current database.

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

Parameter	*logical-DBname*	CSTRING	I	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 DATABASENAME$

SQL_OPEN "TEST1", " "
SQL_EXEC NS_FUNCTION GETCURRENTDBCNTX INTO :DATABASENAME$
;DATABASENAME$="TEST1"

SQL_OPEN "TEST2", " "
SQL_EXEC NS_FUNCTION GETCURRENTDBCNTX INTO :DATABASENAME$
;DATABASENAME$="TEST2"
```

See also SQL_OPEN, SQL_CLOSE, NS_FUNCTION GETDBNAME

NS_FUNCTION GETDBNAME

Retrieves the logical name of the current database.

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

Parameter	*logical-DBname*	CSTRING	O	logical name of the current database

Example

```
Local dbname$

sql_init "NS02MS2K"


if sql_error% < 0
  message 'error' , sql_errmsg$(sql_error%)
endif
sql_open 'pubs', " Support/Support@Venus"
if sql_error% < 0
  message 'open'&&sql_error% , sql_errmsg$(sql_error%)
endif
sql_exec NS_FUNCTION GETDBNAME INTO :dbname$
if sql_error% <> 0
  message 'error' , sql_errmsg$(sql_error%)&&sql_error%
endif
message SQL_GETNAME$, dbname$
; dbname$ contains "Pubs"
```

See also SQL_OPEN, SQL_CLOSE, NS_FUNCTION GETCURRENTDBCNTX

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)	Oerror number	

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
;Récupère le nombre d'erreurs dans ROW_COUNT%
IF ROW_COUNT% <> 0
MOVE 0 TO I%
WHILE i% < ROW_COUNT%
 SQL_EXEC NS_FUNCTION GETERROR :i% INTO :ERROR%
 ;Récupère pour chaque erreur son numéro dans ERROR%
 INSERT AT END "ERROR" && I% && SQL_ERRMSG$(ERROR%) TO LISTBOX1
 I% = I% + 1
ENDWHILE
ENDIF
```

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

NS_FUNCTION GIVECOM

Retrieves in the segment COM_AREA the characteristics of a table whose components are not known at selection.

This function is especially useful when processing dynamic queries and removes the need to define host and FETCH command variables.

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

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

Notes

1. The segment COM_AREA (defined in the file SQL_COM.NCL) is composed of different fields, two of which are pointers (HOST_PTR and SQL_PTR). These two pointers may be retrieved to browse the tables containing the NCL variables (the HOST_PTR pointer) and the SQL variables (the SQL_PTR pointer) concerned by the order being 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)     ;handle towards a segment of NCLELEMENT
                       ;type (defining the NCL host variables)
    int sql_ptr(4)      ;handle towards a segment of SQLELEMENT
                        ;(defining the columns of the query tables)
    int com_ptr(4)      ;reserved
    int num_stat(2)     ;type of queries
                        ; 1 -> SELECT
                        ; 2 -> UPDATE
                        ; 3 -> DELETE
                        ; 4 -> INSERT
                        ; 5 -> others
    int num_col(2)      ; number of columns
    int num_col_compute(2) ;number of COMPUTE columns(not
                           ;applicable for Oracle)
    int len_buf_stat(2) ; size of the buf_stat below
    int buf_stat(4)     ; handle on a buffer containing the
; FETCH INTO instruction [ :,] and as much « :, » as variables to go
; through in a SELECT case
int inited(2)   ;TRUE if it’s OK, FALSE otherwise. To always test
;if it’s TRUE
ENDSEGMENT
```

2. 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.

3. 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.

4. 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) Handle on COM_AREA

Return value INT(4)

```
; Definition of the NCL receiving variables 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) Handle on COM_AREA

Return value INT(4)

```
; Definition of the SQL columns 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 offered 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 the COM_AREA segment..

Variable	*COM_BUFFER%*	INT(4)	Handle on COM_AREA

Return value INT(2)

- **Function SQL_GET_STATEMENT$**

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

The values of the num_stat are the following :

- 1 for SELECT
- 2 for UPDATE
- 3 for DELETE
- 4 for INSERT
- 0 for other type of queries

Variable	*STATEMENT%*	INT(4)	SQL_GET_STATEMENT%

Return value CSTRING

- **Function SQL_GET_NBCOL%**

Returns the number of columns retrieved by the statement.

Variable	*COM_BUFFER%*	INT(4)	Handle on COM_AREA

Return value INT(2)

- **Function SQL_GET_LENGTHFETCH%**

Returns the size of the fetch buffer.

Variable	*COM_BUFFER%*	INT(4)	Handle on COM_AREA

Return value INT(4)

- **Function SQL_GET_FETCHPTR%**

Returns the pointer to the fetch buffer.

Variable	*COM_BUFFER%*	INT(4)	Handle on 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)	Handle on 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)	Handle on 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%

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)	Handle on 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)	Handle on COM_AREA
	COLUMN%	INT(2)	Order of the NCL variable

Return value CSTRING(64)

⚠ Nat System informs you that the five next functions are not very useful with NS_FUNCTION GIVECOM. However, we let them in this documentation for 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%, BUFFER_PTR%
LOCAL COMPUTE% , HEADER$, A$
MOVE "SELECT * FROM EMP" TO A$
SQL_EXECSTR A$
MOVE 0 TO COM_AREA_RET%

WHILE SQL_ERROR% = 0
      SQL_EXEC NS_FUNCTION GIVECOM INTO :COM_AREA_RET%
      IF COM_AREA_RET% = 0
             BREAK
      ENDIF
      INSERT AT END SQL_GET_STATEMENT$ (SQL_GET_STATEMENT%(COM_AREA_RET%) ) TO \
LISTBOX1
```

```
      ; retrieve the command string
      UPDATE LISTBOX1
      IF SQL_GET_STATEMENT%(COM_AREA_RET%) <> 1
             ; the value of the command is different of SELECT
             RETURN 1
      ENDIF

      MOVE SQL_GET_HOSTPTR%(COM_AREA_RET%) TO NCL_PTR%
      ; recuperation of the pointer on the NCL variables array
      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 type column
      IF SQL_GET_LENGTHFETCH%(COM_AREA_RET%) <> 0
             ; if the buffer Fetch size is <> 0
             i% = SQL_GET_FETCHPTR%(COM_AREA_RET%)
             MOV i% , @A$ , 255
      INSERT AT END A$ TO LISTBOX1

             SQL_EXEC_LONGSTR SQL_GET_FETCHPTR%(COM_AREA_RET%), NCL_PTR%, -1
             ; retrieve the pointer to the Fetch buffer + execute
      ELSE
             BREAK
      ENDIF
      WHILE SQL_ERROR% = 0
             MOVE 0 TO I%
             WHILE I% < TOTAL_COL%
             MOVE SQL_GET_HOSTCOLUMNPTR%(COM_AREA_RET% ,i% ) TO BUFFER_PTR%
                    ; retrieve a pointer containing a NCL variable
                    IF BUFFER_PTR% = 0
                           MOVE I% + 1 TO I%
                           CONTINUE
                    ENDIF
                    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
                    EVALUATE SQL_GET_HOSTCOLUMNTYPE%(COM_AREA_RET% ,i% )
                    ; evaluation of the column type
                    ;CONST TYPE_SQL_INT%         0
                    ;CONST TYPE_SQL_STRING%      1
                    ;CONST TYPE_SQL_CSTRING%     2
                    ;CONST TYPE_SQL_NUM%         3
                    ;CONST TYPE_SQL_INSERT_BLOB% 13
                    WHERE 0 ; integer
                           ; retrieve the size of the data
                    EVALUATE SQL_GET_HOSTCOLUMNLENGTH%(COM_AREA_RET% ,i%)
                           WHERE 1 ; 1-byte integer
                           ; retrieve the name of the column
                                  INSERT AT END HEADER$ && \
SQL_GET_SQLCOLUMNNAME$(COM_AREA_RET%, I% ) && ":" &&\
                                                 ASC%
(COM_INT1(BUFFER_PTR%).i1) TO LISTBOX1
                           ENDWHERE
                           WHERE 2 ; it's an integer of 2
                           ; retrieve the name of the column
                           INSERT AT END HEADER$ && \

      SQL_GET_SQLCOLUMNNAME$(COM_AREA_RET% ,i% )&&\
                                                 ":" &&
COM_INT2(BUFFER_PTR%).i2 TO LISTBOX1
                           ENDWHERE
                           WHERE 4 ; it's an integer of 4
                           INSERT AT END HEADER$ && \
      SQL_GET_SQLCOLUMNNAME$(COM_AREA_RET% ,i% )&&\
                                                  ":" &&
COM_INT4(BUFFER_PTR%).i4 TO LISTBOX1
                            ENDWHERE
                            ENDEVALUATE
                            ENDWHERE
                            WHERE 2 ; it's a C string
                            INSERT AT END HEADER$ && \
                            SQL_GET_SQLCOLUMNNAME$(COM_AREA_RET%, I% ) &&\
                            ":" && COM_STRING(BUFFER_PTR%).CS TO LISTBOX1
                            ENDWHERE
                            WHERE 3 ; it's a real
                            EVALUATE SQL_GET_HOSTCOLUMNLENGTH%(COM_AREA_RET%,\
I%)
                            ; retrieve the size of the WHERE 4 ;
                            ; retrieve the name of the column and the value of
                            ; real of size 4
                            INSERT AT END HEADER$ && \
      SQL_GET_SQLCOLUMNNAME$(COM_AREA_RET%, I% ) &&\
      ":" && COM_FLOAT4(BUFFER_PTR%).f4 TO LISTBOX1
                            ENDWHERE
                            WHERE 8 ;
                            ; retrieve the name of the column and the value of
                            ; the real of 8
                            INSERT AT END HEADER$ && \
      SQL_GET_SQLCOLUMNNAME$(COM_AREA_RET%, I% ) &&\
                                                         ":" &&
COM_FLOAT8(BUFFER_PTR%).f8 TO LISTBOX1
                                           ENDWHERE
                                    ENDEVALUATE
                            ENDWHERE
                            ELSE
                            ; retrieve the type of the NCL column
                            INSERT AT END "NCLType" && "INVALID" && \
                            SQL_GET_HOSTCOLUMNTYPE%(COM_AREA_RET%, i%) TO \
LISTBOX1
                            ; retrieve the type of the NCL column
                     ENDEVALUATE
                     MOVE I% + 1 to I%
             ENDWHILE
             SQL_EXEC_LONGSTR SQL_GET_FETCHPTR%(COM_AREA_RET%),NCL_PTR%,-1
             ; execution of the fetch from the pointer on the FETCH buffer
      ENDWHILE
      UPDATE LISTBOX1
      IF SQL_ERROR% = 100
             INSERT AT END "END OF FETCH" TO LISTBOX1
             INSERT AT END "" TO LISTBOX1
      ENDIF
      IF SQL_ERROR% <> 100
             IF SQL_ERROR% > 0
                     MESSAGE "WARNING" && SQL_ERROR% , SQL_ERRMSG$(SQL_ERROR%)
             ENDIF
             IF SQL_ERROR% < 0
                     IF SQL_ERROR% = -32085;No more results to fetch
                        INSERT AT END "END OF RESULT" TO LISTBOX1
                     ELSE
                            MESSAGE "ERROR" &&SQL_ERROR% ,
SQL_ERRMSG$(SQL_ERROR%)
                     ENDIF
             ENDIF
      ENDIF
ENDWHILE
```

See also SQL_EXEC_LONGSTR

NS_FUNCTION IMAGEOFF, IMAGEON

IMAGEON mode enables binary object management (for example bitmaps) of size limited to 32 000 bytes. This handling is carried out in the NCL SEGMENT SQL_IMAGE defined in NSDMS.NCL.

Syntax **NS_FUNCTION IMAGEOFF**

and

NS_FUNCTION IMAGEON

Notes

1. Available from NSnnMS65 in cursor mode 0.
2. IMAGEOFF is the active mode by default..
3. Binary objects are manipulated using an SQL_IMAGE segment:
```
SEGMENT SQL_IMAGE
 INT REALSIZE(4) ; allocation size of the buffer
 INT LENGTH%(4)  ; size really read (when select)
 INT PTR%(4)     ; Address of the buffer
ENDSEGMENT
```
4. To do a SELECT in a binary object, you must be in **DB_MS_CURSORNONE**(2) mode : compatible (cf. NS_FUNCTION SETCURSORMODE).
5. The maximum authorized size is 32K. If you want to handle BLOBs (large images) see SQL_EXEC_LONGSTR (TYPE_SQL_INSERT_BLOB% and TYPE_SQL_SELECT_BLOB%).
6. Images are not the only type of binary objects. Any type of binary file can be stored.
7. ⚠ Binary storage is not cross-platform.

Example

Creation

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

sql_exec create table aelta ( ID INT, LONGSTR  VARCHAR(2000) NULL, COLIMAGE
IMAGE NULL)
if sql_error% <> 0
  message 'error create' , sql_errmsg$(sql_error%)
endif
```

Insertion

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

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


;;;;; To select binaries >  1 Mega, increase text size !!!
Opt$ = 'TEXTSIZE'
Val% = 300001
SQL_EXEC NS_FUNCTION CHANGEOPTION :Opt$, :Val%
if (sql_error% < 0)
      message 'textsize', sql_errmsg$ (sql_error%)
endif

; ---- Changing mode
SQL_EXEC NS_FUNCTION IMAGEON
IF SQL_ERROR% <> 0
      MESSAGE "IMAGEON", SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
      RETURN 1
ENDIF
; ---- Reading of the file and transferring in DATA%
FNAME$ = "(NS-BMP)\TINTIN.BMP"
HBMP%=CREATEBMP%(FNAME$)
BMPT = HBMP%
; twisting the FGETSIZE% bug. Does not interpret the environment variables
FNAME$ = "D:\TESTS\BMP\TINTIN.BMP"

SIZE%=FGETSIZE%(FNAME$) ; = 25000 in this example
INSERT AT END "SIZE "& SIZE% TO LISTBOX1
NEW SIZE%,DATA%
FILE%=F_OPEN%(1,FNAME$)
F_BLOCKREAD FILE%, DATA%, SIZE%, NBREAD%
IF F_ERROR%
      MESSAGE"ERROR", "Failed to load " & FNAME$ &"!"
      F_CLOSE FILE%
      DISPOSE DATA%
      RETURN 1
ENDIF
; ---- Insertion in the t_image table
LOCALIMAGE.REALSIZE = SIZE%
LOCALIMAGE.LENGTH% = SIZE%
LOCALIMAGE.PTR% = DATA%
SQL_EXEC INSERT INTO AELTA (ID, LONGSTR, COLIMAGE) VALUES (1, 'An island \
Between the sky and the water', :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%
SQL_EXEC COMMIT
```

Selection

```
LOCAL DEST$(80),DATA%,SIZE%(4),NBREAD%(2),FILE%,NIL%,FNAME$, hbmp%
LOCAL SQL_IMAGE LOCALIMAGE
LOCAL Opt$
LOCAL Val%
LOCAL CURSORMODE%
; cursor mode none
CURSORMODE% = DB_MS_CURSORNONE ;2
sql_exec NS_FUNCTION SETCURSORMODE :CURSORMODE%

;;;;; To select binaries >  1 Mega, increase text size !!!
```

```
Opt$ = 'TEXTSIZE'
Val% = 3000001
SQL_EXEC NS_FUNCTION CHANGEOPTION :Opt$, :Val%
if (sql_error% < 0)
      message 'textsize', sql_errmsg$ (sql_error%)
endif

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

LOCALIMAGE.realsize = 300000
NEW LOCALIMAGE.realsize,LOCALIMAGE.PTR%
SQL_EXEC SELECT COLIMAGE INTO :LOCALIMAGE FROM AELTA WHERE ID = 1
IF SQL_ERROR% <> 0
      MESSAGE "SELECT IMAGE",SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
ELSE
      ; ---- Displays the image in the CTRLBMP bitmap control
      ; (here LOCALIMAGE.length% values 25000)
      FNAME$="(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%
; ---- return to the default mode
SQL_EXEC NS_FUNCTION IMAGEOFF
; cursor mode default
CURSORMODE% = DB_MS_CURSORDEFAULT ;0
sql_exec NS_FUNCTION SETCURSORMODE :CURSORMODE%
```

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

NS_FUNCTION KILLQUERY

Kills the query sent to the server.

☞ This function has been created to avoid the jamming of an application when a FETCH command take too many times to end.

Syntax **NS_FUNCTION KILLQUERY**

Note

1. This function can be used to abort a query currently being processed by the server.

Example

```
LOCAL I%, PRICEINCLUSIVEOFTAX%, TOTAL%

I%     = 0
TOTAL% = 0
SQL_EXEC SELECT PIOTAX FROM LFACTURE WHERE NOFACT = 10
WHILE SQL_ERROR% = 0
  IF I% >= 4
     SQL_EXEC NS_FUNCTION KILLQUERY
     BREAK
  ELSE
    SQL_EXEC FETCH INTO :PRICEINCLUSIVEOFTAX%
    TOTAL% = TOTAL% + PRICEINCLUSIVEOFTAX%
    I% = I% + 1
  ENDIF
ENDWHILE
MESSAGE " The sum of " & I% & \
        " first lines of the bill n° 10 is", TOTAL%
```

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

Example

```
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
SQL_EXEC SELECT :A$, :B$ FROM :C$ WHERE :A$=:D$
; equals SELECT COL1, COL2 FROM TABLE WHERE COL1 = "HELLO"
...
SQL_EXEC NS_FUNCTION QUOTEON
```

See also NSDBMS.NCL, SQL_ERROR%, SQL_ERRMSG$

NS_FUNCTION ROWCOUNT

Returns the number of rows affected by a query DELETE or UPDATE or the number of FETCH realized after a SELECT.

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

Parameters

nbr-rows	INT(4)	I	number of rows affected by a query

Note

1. The number of rows retrieved by COMPUTE and ROWCOUNT may differ from one row. Indeed, ROWCOUNT can retrieve the number of FETCH realized, whereas COMPUTE retrieve the number of occurences.

Example 1

```
LOCAL ROWCOUNT%

SQL_EXEC DELETE FROM TABPRODUCT\
         WHERE NOPROD >= 30 AND NOPROD < 40

SQL_EXEC NS_FUNCTION ROWCOUNT INTO :ROWCOUNT%
; If 10 recordings correspond to this filter and then 10 recordingss have been
; deleted, thus ROWCOUNT% will contain 10
; If no recording correspond to this filter, thus ROWCOUNT% will contain 0
```

Example 2

```
LOCAL var1%
LOCAL test$
LOCAL ROWCOUNT%
SQL_EXEC SELECT NUM, COL1 FROM BASE
IF SQL_ERROR% <> 0
    MESSAGE "Error ",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%
```

Example 3

```
LOCAL ROWS_AFFECTED%, A$, B$,cursormode%

SQL_EXEC SELECT EMPNO ,ENAME INTO :A$, :B$,:ROWS_AFFECTED% FROM EMP COMPUTE \
COUNT(EMPNO)
IF sql_error% <> 0
  MESSAGE 'error' , sql_errmsg$(sql_error%)&&sql_error%
ENDIF
```

```
WHILE SQL_ERROR%=0 OR SQL_ERROR%=200
 IF SQL_ERROR% =200
  MESSAGE "COMPUTE Number of lines : ", ROWS_AFFECTED%
 ELSE
  INSERT AT END  "ID et NAME"&&A$ && B$ TO LISTBOX1
 ENDIF
 SQL_EXEC FETCH
ENDWHILE

SQL_EXEC NS_FUNCTION ROWCOUNT INTO :ROWS_AFFECTED%

IF SQL_ERROR% <> 0
  MESSAGE 'error' , sql_errmsg$(sql_error%)&&sql_error%
ENDIF

MESSAGE "ROWCOUNT Number of lines : ", ROWS_AFFECTED%
```

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

NS_FUNCTION SETBUFFERSIZE

Specifies the buffer size for the default using mode.

Syntax **NS_FUNCTION SETBUFFERSIZE** *:buffer-size*

Parameter	*buffer-size*	INT(4)	I	size of the buffer

Note

1. Buffer mode and size:

Mode	Maximum size	Minimum size	Default size
Buffering	Defined by the client library in Microsoft SQL Server DBLIB	1	100
Scrolling	Depends on available memory	1	50

Example 1

```
LOCAL curs%
Local int     I%(4)
;Buffering
curs% = 1
sql_exec NS_FUNCTION SETCURSORMODE :curs%
message "cursmode", sql_errmsg$(sql_error%)

Move 100 to I%
SQL_EXEC NS_FUNCTION SETBUFFERSIZE :I%
if sql_error% <> 0
   message 'error' , sql_errmsg$(sql_error%)
endif
```

Example 2

```
LOCAL curs%
Local int I%(4)
;Scrolling
curs% = 2
sql_exec NS_FUNCTION SETCURSORMODE :curs%
message "cursmode", sql_errmsg$(sql_error%)

Move 50 to I%
SQL_EXEC NS_FUNCTION SETBUFFERSIZE :I%
if sql_error% <> 0
   message 'error' , sql_errmsg$(sql_error%)
endif
```

Example 3

See the example of the NS_FUNCTION SETCURSORMODE instruction.

See also NS_FUNCTION SETCURSORMODE, RELATIVE

NS_FUNCTION SETCHARSET

Enables the conversion of a string's character set to another.

Syntax **NS_FUNCTION SETCHARSET** *:charsetin***,** *:charsetout*

Parameters	*charsetin*	INT(4)	I	input character set
	charsetout	INT(4)	I	output character set

Notes

1. Possible values (see NSDBMS.NCL):
   DB_CSET_FIRST 0 /* Default character set identifier */
   DB_CSET_ISO8859_1 DB_CSET_FIRST+ 5 /* ISO 8859-1 */
   DB_CSET_PC850 DB_CSET_FIRST+ 6 /* IBM/OS2 PC850 */
   DB_CSET_PC437 DB_CSET_FIRST+ 7 /* IBM Ascii graphical */
   DB_CSET_HPROMAN8 DB_CSET_FIRST+ 8 /* HP Roman 8 */
   DB_CSET_ROMAN DB_CSET_FIRST+ 9 /* Macintosh character set */
   DB_CSET_EBCDIC_037 DB_CSET_FIRST+10 /* IBM EBCDIC American */
   DB_CSET_EBCDIC_297 DB_CSET_FIRST+11 /* IBM EBCDIC French */
   DB_CSET_EBCDIC_500 DB_CSET_FIRST+12 /* IBM EBCDIC International*/
   DB_CSET_EBCDIC_273 DB_CSET_FIRST+13 /* IBM EBCDIC German */
   DB_CSET_EBCDIC_284 DB_CSET_FIRST+14 /* IBM EBCDIC Spanish */
   DB_CSET_ASCII DB_CSET_PC850 /* IBM/OS2 multi set PC850 */
   DB_CSET_ANSI DB_CSET_ISO8859_1 /* MS/Windows ISO-8859-1 set */
   DB_CSET_LATIN1 DB_CSET_ISO8859_1 /* DEC Latin1 ISO-8859-1 set */
   DB_CSET_ROMAN8 DB_CSET_HPROMAN8 /* HP character set */
2. If charsetIn and charsetOut are the same as DB_CSET_FIRST, no conversion is made.
3. This conversion applies only to STRING and CSTRING data types.
4. The conversion of IN and OUT (INSERT/SELECT) variables is done.

5. If the specified character set does not exist, add the missing character set to the sources supplied with the examples and recompile the conversion DLL (NSxxCSTx.DLL).

6. The SQL string passed as *wSqlExec* is converted.

   Example : `select ... where col1 = 'éùà'`

Example

```
LOCAL CHARSETIN%,CHARSETOUT%
LOCAL I%
LOCAL A$
; conversion from ansi to ascii
CHARSETIN%=DB_CSET_ANSI
CHARSETOUT%=DB_CSET_ASCII
SQL_EXEC NS_FUNCTION SETCHARSET :CHARSETIN%, :CHARSETOUT%
; Activation du module de transaction
IF SQL_ERROR%
 MESSAGE 'SQL ERROR', SQL_ERRMSG$(SQL_ERROR%)
ENDIF
I%=1
A$='èà'
; 'èà' is converted to ascii then stored in the database in ascii char set (IN
; variable)
SQL_EXEC INSERT INTO TOTO (ID, LONGSTR) VALUES (:I%, :A$)
;a$ est stocké en ASCII dans la table (c'est une variable IN donc ANSI -> ASCII)
IF SQL_ERROR%
 MESSAGE 'SQL ERROR', SQL_ERRMSG$(SQL_ERROR%)
ENDIF
SQL_EXEC SELECT ID, LONGSTR FROM TOTO INTO :I%, :A$
IF SQL_ERROR% <> 0
      MESSAGE "Warning SELECT", SQL_ERROR% &&SQL_ERRMSG$(SQL_ERROR%)
endif

; A$ is read in ASCII (it's an OUT variable => ASCII -> ANSI)
INSERT AT END  "ID et LONGSTR"&&I% && A$ TO LISTBOX1

CHARSETIN% =DB_CSET_FIRST
CHARSETOUT% =DB_CSET_FIRST
SQL_EXEC NS_FUNCTION SETCHARSET :CHARSETIN%, :CHARSETOUT%
IF SQL_ERROR%
 MESSAGE 'SQL ERROR', SQL_ERRMSG$(SQL_ERROR%)
ENDIF

; A$ is read in ASCII (No conversion here => ASCII -> ASCII) the result is
; something like '||'
SQL_EXEC SELECT ID, LONGSTR FROM TOTO INTO :I%, :A$
IF SQL_ERROR%
 MESSAGE 'SQL ERROR SELECT', SQL_ERRMSG$(SQL_ERROR%)
ENDIF
INSERT AT END  "ID et LONGSTR"&&I% && A$ TO LISTBOX1
```

See also SQL_ERROR%, SQL_ERRMSG$, NSDBMS.NCL

NS_FUNCTION SETCURSORMODE

Sets the cursor mode used by SQL_OPENCURSOR% or SQL_OPENTHECURSOR% functions.

Syntax **NS_FUNCTION SETCURSORMODE** *:mode*

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

Notes

1. There are three cursor modes: Default, Buffering and Scrolling.
2. Each mode has an associated value and default size:

   For NSnnMSxx :

`DB_MS_CURSORDEFAULT`	Default scrolling	50 rows
`DB_MS_CURSORBUFFER`	Buffering	100 rows
`DB_MS_CURSORNONE`	Compatible	1 row

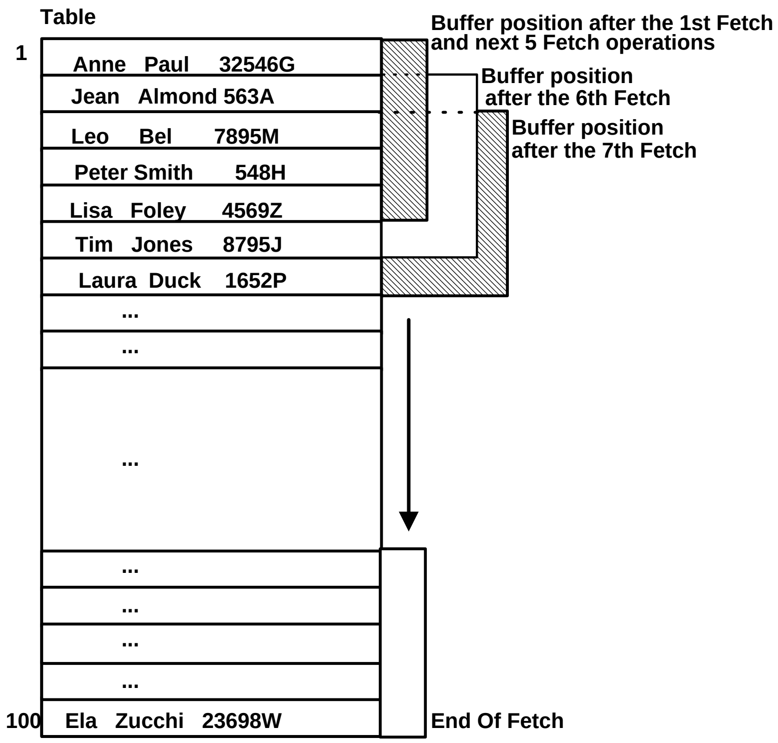
3. The size can be modified by using the command NS_FUNCTION SETBUFFERSIZE.
4. The default mode (since 6.5 version) is DB_MS_CURSORNONE.
5. **Buffering** and **Scrolling** modes store result rows in blocks, which optimizes performance considerably when the results obtained from a SELECT clause are used later for access by relative position.
6. When a table is searched, each FETCH statement is executed individually. You cannot take advantage of the RELATIVE TO positioning function with the **DB_MS_CURSORNONE** mode.

   By default, you can not open multiple cursors with DB_MS_CURSORNONE mode. However if you want to open multiple cursors, execute the following instructions:

   ```
   option$ = 'MULTICURSORS'
   value% = 1
   sql_exec ns_function changeoption :option$, :value%
   ```

7. In **Buffering mode**, when a table is searched, the initial FETCH operations will be executed inside the buffer until its defined size limit is reached. Once the last row in the buffer has been filled, subsequent FETCH operations will be executed individually and will shift the buffer one row at a time until it reaches the required row. In fact, this buffer acts like a stack.

   - Those mode is illustrated in the following diagram. Suppose we have a table with 100 rows and a buffer size of 5.

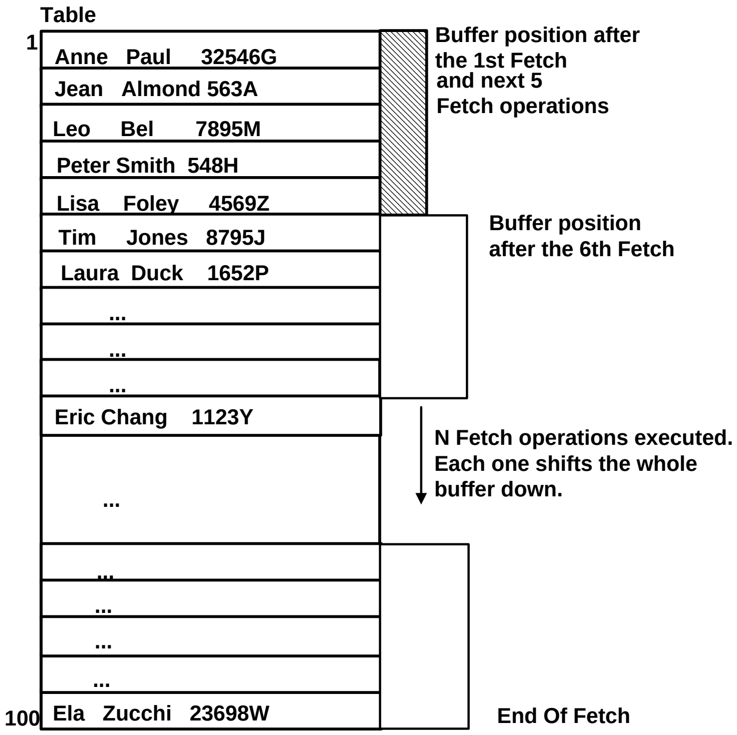


- With **Buffering** mode, you can use the RELATIVE TO command to position the cursor on a row inside the current buffer (but not anywhere in the table).

8. In **Scrolling** mode, when a table is searched, the initial FETCH operations will be executed inside the buffer until its defined size limit is reached. Once the last row in the buffer has been filled, the entire buffer will shift by a distance equal to its size and will contain a new block of rows, which will be affected by subsequent FETCH operations.

- Scrolling mode can only be used on a Microsoft SQL Server server if certain procedures have been installed on the server. If the server configuration does not allow the use of cursor scrolling, the server returns a message indicating that a procedure has not been installed (cf. Microsoft SQL Server server installation).

- This mode is illustrated in the following diagrams. Suppose we have a table with 100 rows and a buffer size of 5.



- With **Scrolling** mode, you can use the RELATIVE TO command to position the cursor on any row in the table (regardless of the current buffer position).

- It also allows you to execute FETCH statements which specify that you want to shift the entire buffer (NEXT and PREVIOUS), position the pointer at the beginning of the first buffer (FIRST) or position the pointer at the beginning of the last buffer (LAST).

- Hence, the syntax for the FETCH statement can be:

  FETCH FIRST INTO : Host_Variable

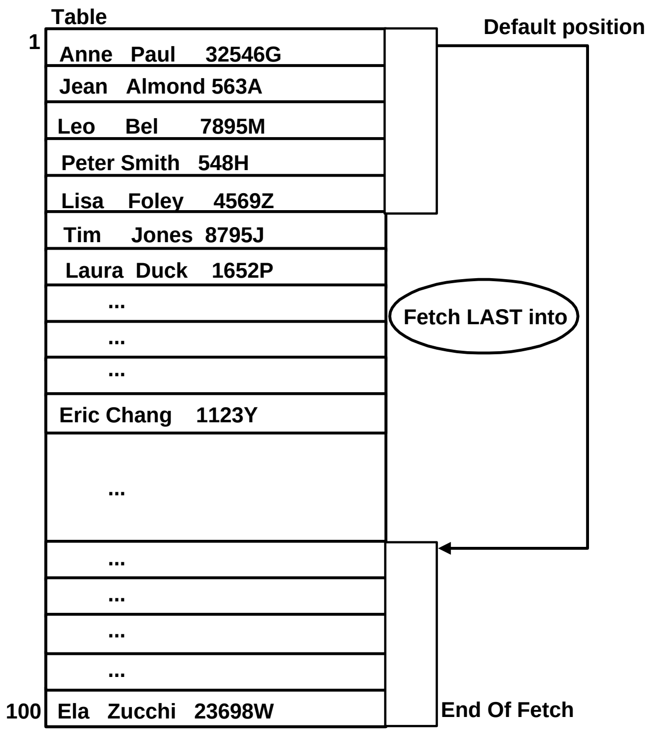
  FETCH LAST INTO : Host_Variable

  FETCH NEXT INTO : Host_Variable

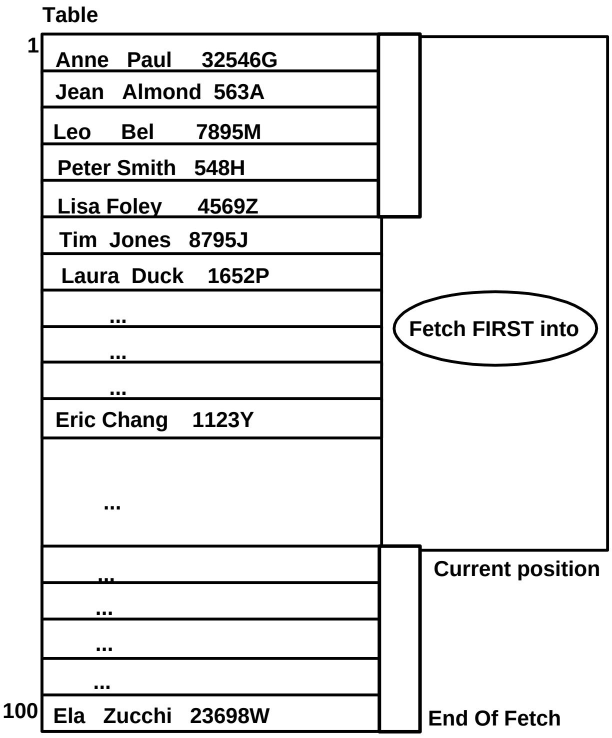
  FETCH PREVIOUS INTO : Host_Variable

9. The diagrams illustrate each of these syntax statements.

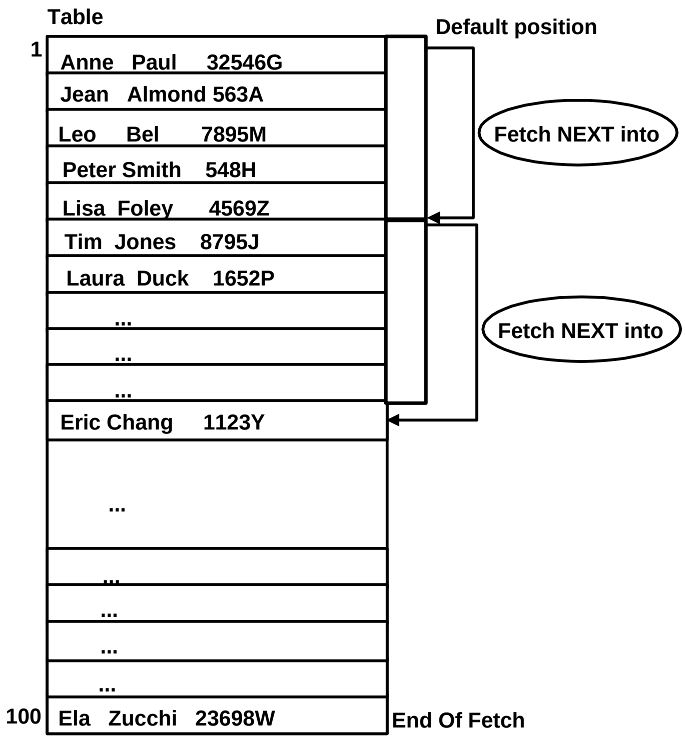
- Example of FETCH LAST into :host_variable



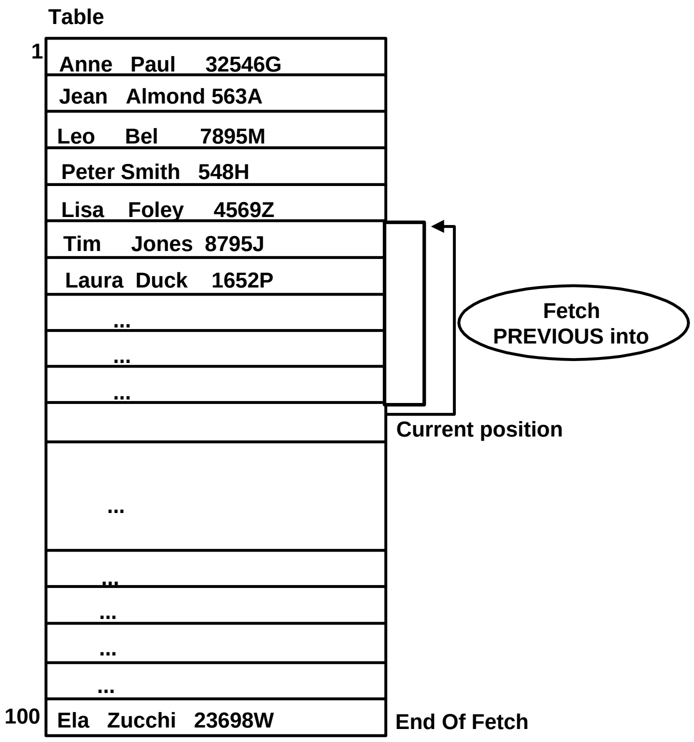
- Example of FETCH FIRST into :host_variable



- Example of FETCH NEXT into :host_variable



- Example of FETCH PREVIOUS INTO :HOST_VARIABLE



10. Relative advantages of these modes:

- The **DB_MS_CURSORNONE** (0) allows you to execute any query but you cannot access rows by relative position.

- **Buffering** mode (1) allows you to fill all the rows in the specified buffer using a single FETCH operation. This is particularly suitable for small selections (<1000 rows).

- **Scrolling** mode (2) allows you to access any row in the table by relative or absolute position, at any time, regardless of the contents of the current buffer.

Example

```
LOCAL MODECURS%

MOVE 1 TO MODECURS%SQL_EXEC SELECT........
SQL_EXEC NS_FUNCTION SETCURSORMODE :MODECURS%

; searches will carry out by buffering mode
SQL_EXEC NS_FUNCTION LOGGINGOFF
```

See also NS_FUNCTION SETBUFFERSIZE, NS_FUNCTION SETCURSORMODE, RELATIVE

NS_FUNCTION SETDEFAULTCURSORMODE

This function allows to set the mode of the cursor opens by default during SQL_OPEN, and open the new cursors in this mode, , without passing via NS_FUNCTION SETCURSORMODE each time.

☞ This function is particularly useful during migration from SQL Server 6.0 to 6.5 or higher.

Syntax **NS_FUNCTION SETDEFAULTCURSORMODE** *:cursor_mode%*

Parameter *cursor_mode%* CSTRING I/O cursor mode

Notes

1. There are three modes of using a cursor: default mode, Buffering mode and Scrolling mode (Scrolling mode does not exist under UNIX).

2. Possible values of the *cursor_mode%* parameter are:

   For NSnnMSxx :

`DB_MS_CURSORDEFAULT`	Default scrolling	50 rows
`DB_MS_CURSORBUFFER`	Buffering	100 rows
`DB_MS_CURSORNONE`	Compatible	1 row

3. The size may be changed using the NS_FUNCTION SETBUFFERSIZE command
4. The **default** mode (since version 6.5) is SCROLLCURSORMODE.
5. **Buffering** mode (corresponding to the value **BUFFERINGCURSORMODE**) and **scrolling** mode (corresponding to the value **SCROLLCURSORMODE**) store by packets of records and achieve considerable optimization in fetching results from a SELECT clause for later use for relative positioning.
6. **OLDCURSORMODE** mode does not benefit from the advantage of RELATIVE TO positioning.
7. SETCURSORMODE sets the cursor mode for the current cursor only. SETDEFAULTCURSORMODE sets the cursor mode for all new cursors.
8. 👓 For the operation of the three modes, refer to NS_FUNCTION SETCURSORMODE.

Example

```
LOCAL name$
LOCAL cursormode%, posSelect%
; scrolling mode
cursormode% = 2
sql_exec NS_FUNCTION SETCURSORMODE :cursormode%
; ---- selection with a lock on select columns
SQL_EXEC SELECT AU_lname INTO :name$ FROM AUTHORS WHERE state = 'CA'
if sql_error% <> 0
  message 'error' , sql_errmsg$(sql_error%)
endif
INSERT AT END  "NAME ="&& name$ TO LISTBOX1
posSelect% = 10
SQL_EXEC RELATIVE TO :posSelect% INTO :name$
if sql_error% <> 0
 message 'error' , sql_errmsg$(sql_error%)
endif
INSERT AT END  "NAME ="&& name$ TO LISTBOX1

; result White Yokomoto
```

See also NS_FUNCTION SETCURSORMODE, NS_FUNCTION SETBUFFERSIZE, RELATIVE

NS_FUNCTION SETDATEFORMAT

NS_FUNCTION SETDATEFORMAT specifies the format in which the driver should return the value of a column of the DATE type during a SELECT..

☞ The default date format is "dd/mm/yyyy H:M:S.L".

Syntax **NS_FUNCTION SETDATEFORMAT** *:format*

Parameter	*format*	CSTRING	I	date format

Notes

1. The default format: d/m/y hmsl
2. Correct formats for dates:
   - D : Day (Jour)
   - M : Month (Mois)
   - Y : Year (An)
3. Valid separators
   - x-x-x
   - x/x/x
   - x.x.x
4. If you include a time format in the character string make sure you separate the date and time formats by a space character.
5. Formats available for the time:
   - H : Hour
   - M : Minute
   - S : Second
   - L : Milliseconds
6. Formats accepted for the time:(without the separators)
   - HMSL
   - HMS
   - HM
   - H
7. If milliseconds are to be included in the command, the seconds should be included in the format before the 'l'. If seconds are to be included, the minutes should be included, etc.

8. The SETDATEFORMATfunction sets the format of the date independently from the date format specified in the configuration panel. So, if the user changes the format in the configuration panel, it has no effect on Nat System development tools.
9. For UNIX platforms, use SETDATEFORMAT2 instead of SETDATEFORMAT.
10. This function is only usable with a SELECT.

Example

```
local mode$, date$
;date format month-year-day hour minute seconds milliseconds
mode$ = 'm-y-d hmsl'
SQL_EXEC NS_FUNCTION SETDATEFORMAT :mode$
IF SQL_ERROR% <> 0
    MESSAGE "Erreur ",SQL_ERRMSG$(SQL_ERROR%)
ENDIF

SQL_EXEC SELECT GETDATE() INTO :date$
IF SQL_ERROR% <> 0
    MESSAGE "Erreur ",SQL_ERRMSG$(SQL_ERROR%)
ENDIF
message  mode$, date$

;date format day/month/year hour minute second
mode$ = 'd/m/y hms'
SQL_EXEC NS_FUNCTION SETDATEFORMAT  :mode$
IF SQL_ERROR% <> 0
    MESSAGE "Erreur ",SQL_ERRMSG$(SQL_ERROR%)
ENDIF

SQL_EXEC SELECT GETDATE() INTO :date$
IF SQL_ERROR% <> 0
    MESSAGE "Erreur ",SQL_ERRMSG$(SQL_ERROR%)
ENDIF
```

NS_FUNCTION SETOEMTOANSI, NS_FUNCTION SETANSITOOEM

Allows the two extended ASCII sets to be used: OEM and ANSI (used by the Windows operating systems).

Syntax

NS_FUNCTION SETOEMTOANSI

and

NS_ FUNCTION SETANSITOOEM

Notes

1. SETOEMTOANSI enables the character set to be converted from extended OEM ASCII to the ANSI set.
2. SETANSITOOEM enables the character set to be converted from extended ANSI ASCII to the OEM set.
3. The client and server for an SQL Server database may be in two different code pages or charset. If this is the situation character data translation should be activated. This option only affects data stored in columns of the type CHAR, VARCHAR and TEXT.
4. For international applications, the server may be in a different code page from the various clients with whom it communicates. So that information should be identical, it is converted to a single code page for storing. For display purposes, the information is reconverted according to the different clients.

Example

```
; a client on ANSI Windows 98 and a server on OEM
; it is impossible to modify the server code page after the installation.
; we have an installed server with 850 code page and a client with the
; option « automatic ANSI to OEM conversion on ON ».
local b$
SQL_EXEC select 'preparation' into :b$
message  "result" , b$
; gives 'preparation' (not Ok)

SQL_EXEC NS_FUNCTION CLEARANSITOOEM
SQL_EXEC select 'preparation' into :b$
message  "result" , b$
; gives 'preparation' (Ok)

;if the server is always with 850 code page but the client is without the option
;« automatic ANSI to OEM conversion on ON »:
;then you must call
SQL_EXEC NS_FUNCTION SETANSITOOEM
```

See also NS_FUNCTION CLEAROEMTOANSI, NS_FUNCTION CLEARANSITOOEM

NS_FUNCTION CLEAROEMTOANSI, NS_FUNCTION CLEARANSITOOEM

Enables removal of the conversion of the extended OEM or ANSI ASCII character set.

Syntax **NS_FUNCTION CLEAROEMTOANSI**

and

NS_ FUNCTION CLEARANSITOOEM

Notes

1. CLEAROEMTOANSI clears conversion of the extended OEM ASCII character set into ANSI.
2. CLEAROEMTOOEM clears conversion of the extended ANSI ASCII character set into OEM.

See also NS_FUNCTION SETOEMTOANSI, NS_FUNCTION SETANSITOOEM

NS_FUNCTION STATEMENT

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

Syntax **NS_FUNCTION STATEMENT INTO** : *query-string*

Parameters			
query-string	CSTRING	I/O	statement used in the query sent to the SQL engine

Note

1. In **Scrolling mode**, the statement returned may have been modified by Microsoft SQL Server.
2. This functionality is not usable in **SCROLLCURSORMODE** mode.
3. The INTO clause (even precised) is never traced.

Example

```
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 query is :", PHRASE$

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

See also NS_FUNCTION SETCURSORMODE, NS_FUNCTION SETBUFFERSIZE

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 CHAR. TRIMCHARON is available only with host variables of CSTRING, STRING type, but not in CHAR or VARCHAR2 type.

Syntax **NS_FUNCTION TRIMCHAROFF**

and

NS_FUNCTION TRIMCHARON

Notes

1. TRIMCHAROFF is the default mode.
2. TRIMCHARON is of limited importance with SQL Server. SQL Server does not insert blanks at the end of a string. But TRIMCHARON may be useful if blanks have been inserted by a means external to Nat System development products.

Example

```
LOCAL C$
SQL_EXEC CREATE TABLE T_DEMO(TEST CHAR(10))

SQL_EXEC INSERT INTO T_DEMO(TEST) VALUES ("A234567890")
SQL_EXEC INSERT INTO T_DEMO(TEST) VALUES ("A2345")
SQL_EXEC INSERT INTO T_DEMO(TEST) VALUES ("A")

; this is the default mode
; ----- this loop displays <A234567890>
;                              <A2345     >
;                              <A         >
SQL_EXEC SELECT * FROM T_DEMO
WHILE SQL_ERROR% <> 0
  SQL_EXEC FETCH INTO :C$
  MESSAGE "C$=<" & C$ & ">",""
ENDWHILE

; ----- Changing mode
SQL_EXEC NS_FUNCTION TRIMCHARON

; ----- this loop will display <A234567890>
;                              <A2345>
;                              <A>
SQL_EXEC SELECT * FROM T_DEMO
WHILE SQL_ERROR% <> 0
  SQL_EXEC FETCH INTO :C$
  MESSAGE "C$=<" & C$ & ">",""
ENDWHILE

; ---- Return to the default mode
SQL_EXEC NS_FUNCTION TRIMCHAROFF
```

See also SQL_ERROR%, SQL_ERRMSG$