NatStar

Version 5.00 Edition 1

NS-DK

Version 5.00 Edition 1

NatWeb

Version 4.00 Edition 1

Common Database Access Interface

Ref. n° NS500COMMU00033

Contents

About this Manual

This manual describes the common functions and instructions of all databases access interfaces.

Supported configurations

Development environment

- Windows 16 bits : 3.11, 95, 98, NT 4.0
- Windows 32 bits : 95, 98, NT 4.0, 2000
- OS/2 16 and 32 bits

Client environment

Operating system	**DBMS drivers available**	**Communication's services**
OS/2 16 et 32 bits	Oracle 7.3 DB2 2.1, 5.0 Sybase 10 Informix 5.01	NatStar/TP/E
Windows 16 bits	ODBC 2.0 Oracle 7.3 DB2 2.1, 5.0 MS SQLServer 6.5 Sybase 10, 11 Informix 5.01	Tuxedo 6.5 NatStar/TP/E
Windows 32 bits	ODBC 3.0 Oracle 8.0, 8.1 DB2 2.1, 5.0, 6.1 MS SQLServer 7.0 MS SQLServer 2000 Sybase 11, 12 Informix 7.2, 9.2	Tuxedo 6.5, 7.1 NatStar/TP/E

Server environment

Operating system	**DBMS drivers available**	**Communication's services**
Windows NT Windows 2000 32 bits	ODBC 3.0 Oracle 8.0, 8.0XA Oracle 8.1, 8.1XA DB2 2.1, 5.0, 6.1 MS SQLServer 7.0 MS SQLServer 2000 Sybase 11, 11XA Sybase 12, 12XA Informix 7.2, 9.22	Tuxedo 6.5, 7.1 NatStar/TP/E
AIX 4.1	Oracle 8.0, 8.0 XA	Tuxedo 6.5

Operating system	**DBMS drivers available**	**Communication's services**
32bits	Sybase 11 Informix 7.2, 7.2 XA	NatStar/TP/E
AIX 4.3 32bits	Oracle 8.0, 8.0 XA Oracle 8.1, 8.1 XA Sybase 11 Informix 7.2, 7.2 XA Informix 9.22, 9.22 XA	Tuxedo 6.5, 7.1 NatStar/TP/E
HP-UX 10.x 32 bits	Oracle 8.0, 8.0 XA Sybase 11 Informix 7.2, 7.2 XA	Tuxedo 6.5 NatStar/TP/E
HP-UX 11.x (Risc 2) 32 bits	Oracle 8.0, 8.0 XA Oracle 8.1, 8.1 XA Sybase 11 Informix 7.2, 7.2 XA Informix 9.22, 9.22 XA	Tuxedo 6.5, 7.1 NatStar/TP/E
Sun Solaris 2.5 32 bits	Oracle 8.0, 8.0 XA Sybase 11 Informix 7.2, 7.2 XA	Tuxedo 6.5 NatStar/TP/E
Sun Solaris 2.7 32 bits	Oracle 8.0, 8.0 XA Oracle 8.1, 8.1 XA Sybase 11 Informix 7.2, 7.2 XA Informix 9.22, 9.22 XA	Tuxedo 6.5, 7.1 NatStar/TP/E
Linux RedHat 6.2	Oracle 8.1, 8.1 XA	Tuxedo 7.1 NatStar/TP/E
MVS/CICS/IMS MVS/Batch	DB2	NatStar/TP/E CICS
AS400	DB2	Tuxedo 6.5, 7.1 NatStar/TP/E

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 all interfaces.

Chapter 1	**Common Database Access Interface** This chapter describes the APIs common among the drivers.

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)
16	Indicates specific information on using the software under DOS- Windows Windows 3.x (16 bits)
32	Indicates specific information on using the software under DOS- Windows 32 bits
OS/2	Indicates specific information on using the software under OS/2- PM (all versions)
OS/2 16	Indicates specific information on using the software under OS/2- PM version 1.3 and later (16 bits)
OS/2 32	Indicates specific information on using the software under OS/2- PM version 2.x (32 bits)
16 bits	Indicates specific information on using the software under OS/2- PM 1.x or DOS- Windows 3.x (16 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 Common Database Access Interface

The chapter describes the common interface of all databases accessed with the Nat System development tools..

This chapter explains

- The components of the DBMS, arranged in functional categories.
- The reference of the components in these libraries.
- The reference of the NS_FUNCTION extensions in the libraries.

Contents

Introduction

This chapter describes the APIs common among the drivers. They are grouped together by functional categories (loading drivers, error handling, ...). Detailed information about specific functions of a DBMS is found in its respective chapter.

For example, this chapter describes error handling in general. But error messages, specific to each driver, are described in the chapter dealing with the DBMS.

Functional categories

Here is a list, arranged by functional category, of the instructions, functions and constants common to all databases.

Selecting the DBMS driver to use

Executing SQL commands

Mechanisms to execute an SQL order

Managing cursors

There are two categories of cursors:

The first category is cursors which are managed like LIFO (Last In First Out) stacks. They use the following APIs:

```
c1%=SQL_OPENCURSOR%
SQL_CLOSECURSOR%
```

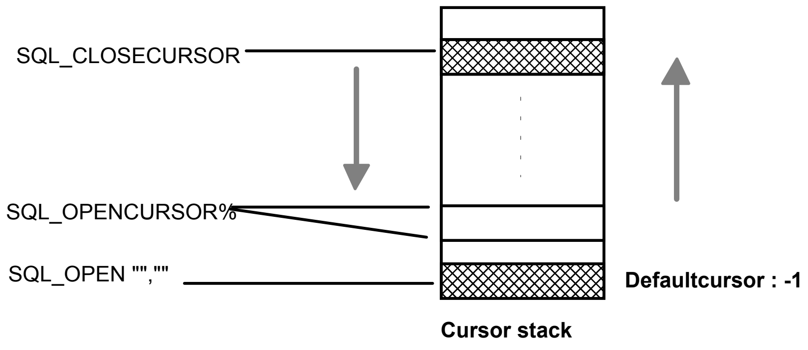
The second category is cursors which allow the explicit closing of a given cursor. They use the following APIs:

```
c2%=SQL_OPENTHECURSOR%
SQL_CLOSETHECURSOR(c2%)
```

Even though both modes can be in the same application, SQL_ERROR% will return an error if you try to execute the following commands:

SQL_CLOSETHECURSOR when the cursor was opened with SQL_OPENCURSOR%, SQL_CLOSECURSOR when the cursor was opened with SQL_OPENTHECURSOR%.

Managing cursor stack



Managing cursors with explicit cursor closing

Managing errors

Uncentralized error management

The following functions apply to all DBMS drivers.

Centralized error management

This mode of error management gives you more control or error management and is more powerful than the functions of SQL_ERROR%, SQL_ERRMSG$, NS_FUNCTION ERRORCOUNT and NS_FUNCTION GETERROR.

Library reference

SQL_INIT Instruction

Loads the driver needed to use a DBMS for a given target.

Syntax **SQL_INIT** *DLL-name*

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

Notes

1. This must be the first SQL_ instruction called by any application that wants to use a DBMS with NCL.
2. The *DLL-name* parameter should contain the name of the DLL used to access the DBMS.
3. For the *DLL-name* you use for your DBMS, see the chapter dedicated to the DBMS.

Example

```
; ---- Example for ORACLE 8 ;0
SQL_INIT "NSW2OR8" ; load ORACLE 8.0 driver
IF SQL_ERROR% <> 0
   MESSAGE "Error loading DLL",SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
   RETURN
ENDIF
...
SQL_STOP ; Unload the driver
```

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

SQL_STOP instruction

Unloads the current DBMS driver and closes all open databases and cursors.

Syntax **SQL_STOP**

Note

1. 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. The *logical-DBname* parameter specifies the logical database name
2. The *connection-string* parameter specifies the command string used to connect to a local or remote database.

Example

See example of SQL_CLOSE instruction.

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

```
; ---- Exemple ODBC2

; ----  DATASOURCE DSSYBASE connect
SQL_OPEN "BASE1", "usr1/pswd1@DSSYBASE"
IF SQL_ERROR% <> 0
   MESSAGE "Erreur Base1", SQL_ERRMSG$(SQL_ERROR%)
ENDIF
...
; ---- DATASOURCE DSSYBASE disconnect
SQL_CLOSE "BASE1"
```

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

```
; ---- Example for RDB
SQL_OPEN "BASE1"  , "USR1/PWD1@NODENAME1#DECNET"
SQL_OPEN "CLASS2" , "USR1/PWD1@NODENAME1#DECNET!SERVERCLASS"
SQL_OPEN "BASE3"  , "USR1/PWD1@NODENAME1"

SQL_EXEC SELECT...            ; SELECT sur BASE3

SQL_EXEC AT CLASS2 SELECT... ; SELECT sur CLASS2
SQL_EXEC AT CLASS2 FETCH...  ; FETCH  sur CLASS2

SQL_EXEC FETCH...             ; FETCH sur BASE3
```

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

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

```
; ---- Example with SYBASE 11
LOCAL LOGICALDBNAME$

; ---- Opening of the logical database BASE1 associated to a connection on the
;       logical server SERV1 with the physical database B1
SQL_OPEN "BASE1","USR1/PSWD1@SERV1"
SQL_EXEC USE B1 ; SQL order of SYBASE because BASE1 is not
                ; the name of the physical database of SERV1
SQL_EXEC ...    ; BASE1 is the current database

; ---- Opening of the logical database pubs2
;       associated to a connection on the
;       logical server SERV1 with the physical database pubs2
SQL_OPEN "PUBS2/NOCURSOR","USR1/PSWD1@SERV1"
SQL_EXEC ...    ; pubs2 is the current database

LOGICALDBNAME$ = "BASE1"
SQL_EXEC NS_FUNCTION CHANGEDBCNTX :LOGICALDBNAME$
SQL_EXEC ...    ; BASE1 is the current database

SQL_CLOSE "BASE1"
SQL_EXEC ...    ; pubs2 is the current database

SQL_CLOSE "pubs2"
```

See also SQL_OPEN, SQL_CLOSE, NS_FUNCTION GETDBNAME, AT

SQL_EXEC Instruction

Executes an SQL command.

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

Parameters

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

Notes

1. The SQL command is passed directly without quotes. It can correspond to any Oracle SQL command , whether it's a data definition command (CREATE TABLE, CREATE INDEX, ...) or a data manipulation command (SELECT, INSERT, UPDATE, ...).

2. The AT command can only be used with databases which allow several simultaneous connections. The query is sent to the database specified after the AT command (without quotes and case-sensitive). If the AT command isn't specified, the SQL_EXEC executes on the current database.

3. If USING cursor_handle is specified, it indicates which cursor previously opened by SQL_OPENCURSOR% must be used to execute the SQL command. If no cursor has been opened, the cursor's value is that of DEFAULT_CURSOR: -1.

4. The SQL command can return values in NCL variables. For this, just pass these variables in parameters.

5. It is possible to pass a segment's field as a data-receiving variable in an SQL query.

6. The commands SQL_EXEC, SQL_EXECSTR and SQL_EXEC_LONGSTR depend on the SQL language accepted by the DBMS in use (Refer to the DBMS documentation).

7. For SQL commands that are too long, it is possible to use the special continuation character "\":

```
SQL_EXEC UPDAPTE SAMPLE SET COMPANY  =:A$ \
                       WHERE TOWN    =:C$ AND \
                             COUNTRY =:D$
```

8. The types of variables recognized by the interface are:

INT(1), INT(2), and INT(4)

NUM(8), NUM(4)

STRING

CSTRING

CHAR

9. Each database has its own implementation of SQL. 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 a table.

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

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

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

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

Example

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

CITY$ = "NEW YORK"

; ========
;  1st example
; ========
; ---- Select a subset
SQL_EXEC SELECT CODE,COUNTRY FROM WORLD WHERE TOWN =:CITY$

; ---- Read the first to last entry
WHILE SQL_ERROR% = 0
  SQL_EXEC FETCH INTO :CODE%,:COUNTRY$
  IF SQL_ERROR% = 0
     INSERT AT END CODE% && COUNTRY$ TO LBOX1
  ENDIF
ENDWHILE

; =============================
;  2nd example(most efficient)
; =============================
; ---- Select a subset
;      and read the first entry
SQL_EXEC SELECT CODE,COUNTRY FROM WORLD INTO:CODE%,:COUNTRY$ WHERE TOWN =:CITY$

; ---- Read the second to the last entry
WHILE SQL_ERROR% = 0
  INSERT AT END CODE% && COUNTRY$ TO LBOX1
  SQL_EXEC FETCH
ENDWHILE

; =========
;  3rd example
; =========
; ---- Select a subset
SQL_EXEC SELECT CODE,COUNTRY FROM WORLD WHERE TOWN =:CITY$

; ---- Read 1st entry to last entry
;      by filling TCODE% and TCOUNTRY$ tables
I% = 0
WHILE (SQL_ERROR% = 0) AND (I% < 10)
  SQL_EXEC FETCH INTO :TCODE%[I%],:TCOUNTRY$[I%]
  I% = I% + 1
ENDWHILE


; =============================
;  Using flags
; =============================
SQL_EXEC CREATE TABLE FAMILY( NAME       VARCHAR2(10),\
                              AGE        NUMBER,      \
                              CHILDNAME  VARCHAR2(10))
FATHER$ = "STEVE"
AGE%    = 35
SON$    = "PETER"
IND1%   = 0
IND2%   = 0
```

```
; --- Insert "STEVE",35,"PETER" into table
SQL_EXEC INSERT INTO FAMILY VALUES (:FATHER$:IND1%, :AGE%, :SON$:IND2%)

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

; ---- The SELECT loop places the listbox LBOX
;       'STEVE's son is PETER'
;       'PETER does not have a son.'
SQL_EXEC SELECT NAME, AGE, CHILDNAME INTO: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 Nat System DLL specifically designed for the DBMS stores cursors in a LIFO (Last In First Out) stack: SQL_OPENCURSOR% stacks and SQL_CLOSECURSOR unstacks.

8. The following rules apply when executing a statement with a cursor:

   - Statements are always executed with the specified cursor.

- If with SQL_EXEC, the USING clause isn't specified, the commands are executed with the DEFAULT_CURSOR.

9. When several databases are opened simultaneously, the cursor opened by SQL_OPENCURSOR% is immediately associated with the current database.

10. If you want to open a cursor in a database other than the current one, you must execute the SQL_EXEC CHANGEDBCNTX:otherbase$ command to change databases before you execute SQL_OPENCURSOR%.

Example

See the example of SQL_CLOSETHECURSOR instruction.

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

SQL_CLOSECURSOR instruction

Closes the last cursor opened

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 can only close cursors opened with SQL_OPENCURSOR%.

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%**

Return value INT(4)

Notes

1. After opening the cursor, it can be used with the following instructions:

```
SQL_EXEC SELECT ... USING cursor-handle
SQL_EXEC FETCH ... USING cursor-handle
```

2. A cursor is an internal resource managed by the Nat Systems 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 an 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 database other than the current one, you must execute the SQL_EXEC CHANGEDBCNTX:otherbase$ command to change databases before you execute SQL_OPENCURSOR%.

Example

See the example of the SQL_CLOSETHECURSOR instruction.

See also SQL_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*

Parameter *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"
```

Example

```
MOVE "SAMPLE" TO B$
SQL_OPEN B$, ""
IF SQL_ERROR% < 0
   MESSAGE "fatal error on" && B$, SQL_ERRMSG$(SQL_ERROR%)
   MESSAGE "Danger !", "Application stopped"
   RETURN
ELSEIF SQL_ERROR% > 0
       MESSAGE "Warning on" && B$, SQL_ERRMSG$(SQL_ERROR%)
    ELSE
       MESSAGE "OK", "Base" && B$ && "opened"
ENDIF
```

See also 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 SQL_ERROR%, NS_FUNCTION ERRORCOUNT, NS_FUNCTION GETERROR, NS_FUNCTION CALLBACK

NS_FUNCTION ERRORCOUNT

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

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

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

Example

```
LOCAL NBERROR%

SQL_EXEC NS_FUNCTION ERRORCOUNT INTO :NBERROR%
MESSAGE "NUMBER OF ERRORS",NBERROR%
```

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

NS_FUNCTION GETERROR

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

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

Parameters

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

Example

```
LOCAL I%, ROW_COUNT%, ERROR%

MOVE 0 TO ROW_COUNT%

SQL_EXEC NS_FUNCTION ERRORCOUNT INTO :ROW_COUNT%
 ;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%
             MESSAGE "ERROR" && I%, SQL_ERRMSG$(ERROR%)
             I% = I% + 1
      ENDWHILE
ENDIF
```

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

NS_FUNCTION CALLBACK

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
6. 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.
7. The type of the error's message is sent to PARAM12%. The handle of the structure is sent to PARAM34%.

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

SQL_GETNAME$ function

Returns the name of the corresponding library for the DBMS currently in use.

Syntax **SQL_GETNAME$**

Returned value CSTRING

Example

```
LOCAL HOR8%, HS11%

; --- load NSw2OR8
HOR8% = SQL_INITMULTIPLE% ("NSw2OR8")

; --- load NSw2S11
HS11% = SQL_INITMULTIPLE% ("NSw2S11")

MESSAGE "Library corresponding to the current DBMS", SQL_GETNAME$
; return NSw2S11

SQL_USE HINF%
; --- load the driver
MESSAGE "Library corresponding to the current DBMS", SQL_GETNAME$
; return NSw2OR8
```

See also SQL_PRODUCT$, SQL_VERSION$, SQL_USE, SQL_ERROR%, SQL_ERRMSG$

SQL_GETTIME% function

Returns the time taken by the last SQL command executed (in milliseconds).

Syntax **SQL_GETTIME%**

Returned value INT(4)

Example

```
LOCAL HOR8%

HOR8% = SQL_INITMULTIPLE% ("NSw2OR8")

SQL_OPEN "base", ""
MESSAGE "Open took", SQL_GETTIME% && "ms"

SQL_EXEC SELECT ...
MESSAGE "select took", SQL_GETTIME% && "ms"
```

See also SQL_STARTTIMER, SQL_STOPTIMER, SQL_GETTIMER%, SQL_ERROR%, SQL_ERRMSG$

SQL_STARTTIMER instruction

Starts the timer.

Syntax	**SQL_STARTTIMER**
Example	
	See the example of the SQL_GETTIMER% instruction.
See also	SQL_STOPTIMER, SQL_GETTIMER%, SQL_ERROR%, SQL_ERRMSG$

SQL_STOPTIMER instruction

Stops the timer.

Syntax	**SQL_STOPTIMER**
Example	
👓	See the example of the SQL_GETTIMER% instruction.
See also	SQL_STARTTIMER, SQL_GETTIMER%, SQL_ERROR%, SQL_ERRMSG$

SQL_GETTIMER% function

Depending on its parameter, returns the SQL, NCL, or combined (SQL+NCL) time of execution of SQL commands between the STARTTIMER and STOPTIMER instructions.

Syntax	**SQL_GETTIMER%**(*timertype*)			
Parameter	*timertype*	INT(2)	I	TIMER type
Returned value	INT(4)			

Notes

1. *Timertype* is an INT(2) and can have the following values:
   - CONST SQL_TIME
   - CONST NCL_TIME
   - CONST GLOB_TIME
2. If *timertype* is SQL_TIME, only the SQL time is returned.
3. If *timertype* is NCL_TIME, only the NCL time is returned.
4. If *timertype* is GLOB_TIME, returns the combined time (SQL + NCL.)

Example

```
...
SQL_STARTTIMER
SQL_EXEC SELECT COL1,COL2 INTO  :A$,  :B$ FROM TABLE1
WHILE SQL_ERROR% = 0
  SQL_EXEC FETCH
  INSERT AT END A$,B$ TO LBOX
ENDWHILE
SQL_STOPTIMER
INSERT AT END "TEMPS SQL  =" & SQL_GETTIMER%(SQL_TIME)   TO LBOX
INSERT AT END "TEMPS NCL  =" & SQL_GETTIMER%(SQL_NCL)    TO LBOX
INSERT AT END "TEMPS TOTAL=" & SQL_GETTIMER%(SQL_GLOBAL) TO LBOX
...
```

See also SQL_STARTTIMER, SQL_STOPTIMER, SQL_ERROR%, SQL_ERRMSG$

SQL_GETUSED% function

Returns the handle of the DBMS currently in use.

Syntax **SQL_GETUSED%**

Returned value INT(2)

Example

```
LOCAL HOR8%, HS11%, H%

HOR8% = SQL_INITMULTIPLE% ("NSw2OR8")

HS11% = SQL_INITMULTIPLE% ("NSw2S11")

H% = SQL_GETUSED%
; Value of h% is equivalent to hS10%
SQL_STOPMULTIPLE H%
; Close NSw2S11
```

See also SQL_USE, SQL_ERROR%, SQL_ERRMSG$

SQL_INITMULTIPLE% function

Defines a DBMS and initializes it.

Syntax **SQL_INITMULTIPLE%** *(DBMS-name)*

Parameters	*DBMS-name*	CSTRING	I	name of the corresponding library for the DBMS used.

Returned value INT(2)

Notes

1. This must be the first function called by any application that wants to use a DBMS (it is responsible for loading the library).
2. This function has the same effect as SQL_INIT but can be called several times with different DBMS names; it allows applications to work with several DBMSs at the same time.
3. The function returns a handle that uniquely identifies the DBMS
4. ⚠ This function cannot be used together with SQL_INIT in a program. Before writing a program, developers need to decide whether they are working in multi-DBMS mode or single DBMS mode. Similarly, the functions SQL_STOPMULTIPLE, SQL_USE and SQL_GETUSED% can only be used in a multi-DBMS context.

Example

```
LOCAL HOR8%, HS11%

; ---- Load library for ORACLE 8
HOR8% = SQL_INITMULTIPLE% ("NSw2OR8")
IF SQL_ERROR% <> 0
  MESSAGE "Error of ORACLE's loading", SQL_ERRMSG$ (SQL_ERROR%)
ENDIF

LOCAL HOR8%, HS11%

; ---- Load library for ORACLE 8
HOR8% = SQL_INITMULTIPLE% ("NSw2OR8")
IF SQL_ERROR% <> 0
  MESSAGE " Error of ORACLE's loading", SQL_ERRMSG$ (SQL_ERROR%)
ENDIF

; ---- Load library for SYBASE 11 (is now the current driver)
HS11% = SQL_INITMULTIPLE% ("NSw2S11")
IF SQL_ERROR% <> 0
  MESSAGE " Error of Sybase's loading ", SQL_ERRMSG$ (SQL_ERROR%)
ENDIF
```

```
; ---- Connect to SYBAXE 11 pubs2 database
SQL_OPEN "PUBS2","USR1/PWSD1@SERV1"

; ---- Change current driver
SQL_USE HOR8%

; ---- Connect to ORACLE 8 using the service COMPTA1
SQL_OPEN "BASE1", "SCOTT/TIGER@COMPTA1"

; ---- Unload all loaded libraries
SQL_STOPALL
```

See also SQL_INIT, SQL_STOP, SQL_STOPMULTIPLE, SQL_ERROR%, SQL_ERRMSG$

SQL_LOGGINGON instruction

Starts automatically logging all queries executed by an application.

Syntax	**SQL_LOGGINGON** *log-filename*			
Parameters	*log-filename*	CSTRING	I	name of the file used to store the log

Notes

1. If you want to log everything, this must be the first instruction.
2. If the file already exists, it will be reinitialized when this function is executed, otherwise it will be created.
3. Logging will continue until the SQL_LOGGINGOFF instruction is encountered (or the application terminates).

Example

```
SQL_LOGGINGON "C:\MATRACE.TRC"

SQL_INIT "NSw2DB26"
SQL_OPEN "MYBASE", ""

SQL_LOGGINGOFF
```

See also SQL_LOGGINGOFF, SQL_ERROR%, SQL_ERRMSG$

SQL_LOGGINGOFF instruction

Terminates the current log.

Syntax **SQL_LOGGINGOFF**

Example

```
SQL_LOGGINGON "C:\MATRACE.TRC"

SQL_INIT "NSw2S11"
SQL_OPEN "MYBASE", ""

SQL_LOGGINGOFF
```

See also SQL_LOGGINGON, SQL_ERROR%, SQL_ERRMSG$

SQL_PRODUCT$ function

Returns the full name of the DBMS associated with the library currently in use.

Syntax **SQL_PRODUCT$**

Returned value CSTRING

Notes

1. Nat System invites you to use this function and SQL_VERSION$ function before to call the technical support, because these two functions allow to identify very precisely the version and the driver used.

Example

```
LOCAL HOR8%, HS11%

HOR8% = SQL_INITMULTIPLE% ("NSw2OR8")

HS11% = SQL_INITMULTIPLE% ("NSw2S11")

MESSAGE "SGBD en cours d'utilisation", SQL_PRODUCT$
; returns the product name for NS02S10

SQL_USE HOR8%
MESSAGE "SGBD en cours d'utilisation", SQL_PRODUCT$
; returns the product name for NS02OR8
```

See also SQL_GETNAME$, SQL_VERSION$, SQL_ERROR%, SQL_ERRMSG$

SQL_STOPALL instruction

Terminates all initialized DBMSs.

Syntax **SQL_STOPALL**

Example

```
LOCAL HOR8%, HS11%

HOR8% = SQL_INITMULTIPLE% ("NSw2OR8")
...
HS11% = SQL_INITMULTIPLE% ("NSw2S11")
...
SQL_STOPALL
```

See also SQL_STOPMULTIPLE, SQL_STOP, SQL_ERROR%, SQL_ERRMSG$

SQL_STOPMULTIPLE instruction

Terminates use of a DBMS.

Syntax **SQL_STOPMULTIPLE** *handle*

Parameters

handle	INT(2)	I	handle of the DBMS to terminate

Note

1. Can only be used with DBMSs initialized with SQL_INITMULTIPLE%.

Example

```
LOCAL HOR8%,HS11%
HOR8% = SQL_INITMULTIPLE% ('NSw2OR8')
HS11% = SQL_INITMULTIPLE% ('NSw2S11')
...
SQL_STOPMULTIPLE HOR8%
IF SQL_ERROR% <> 0
MESSAGE " Error unloading library ", SQL_ERRMSG$ (SQL_ERROR%)
ENDIF

SQL_STOPMULTIPLE HS11%
IF SQL_ERROR% <> 0
MESSAGE " Error unloading library ", SQL_ERRMSG$ (SQL_ERROR%)
ENDIF
```

See also SQL_INITMULTIPLE%, SQL_STOP, SQL_INIT, SQL_ERROR%, SQL_ERRMSG$

SQL_USE instruction

Selects the DBMS used by any subsequent SQL statements.

Syntax **SQL_USE** *handle%*

Parameters	*handle%*	INT(2)	I	handle of a DBMS initialized earlier by SQL_INITMULTIPLE%

Note

1. Any subsequent SQL statements will be applied to the selected DBMS until the next SQL_USE instruction.

Example

```
LOCAL HOR8%, HS11%

HOR8% = SQL_INITMULTIPLE% ("NSw2OR8")
...
HS11% = SQL_INITMULTIPLE% ("NSw2S11")
SQL_USE HOR8%
SQL_OPEN "BASE1", ""
...
SQL_USE HS11%
SQL_OPEN "BASE2", ""
```

See also SQL_INITMULTIPLE%, SQL_GETUSED%, SQL_ERROR%, SQL_ERRMSG$

SQL_VERSION$ function

Returns the version of the library for the DBMS currently in use.

Syntax **SQL_VERSION$**

Notes

1. Nat System invites you to use this function and SQL_PRODUCT$ function before to call the technical support, because these two functions allow to identify very precisely the version and the driver used.

Returned value CSTRING

Example

```
LOCAL HOR8%
HOR8% = SQL_INITMULTIPLE% ("NSW2OR8")
MESSAGE "Version", SQL_VERSION$
; Returns "ORACLE WINDOWS NT / DLL 2.0 / ORA8 PROD.9 / Sep 20 1996"
...
```

See also SQL_PRODUCT$, SQL_GETNAME$, SQL_ERROR%, SQL_ERRMSG$

Error codes

These error codes are used internally by the NSnn_SQL library:

+100 ** NO ROW WAS FOUND OR LAST ROW REACHED
Cause: End of search sequence retrieved by the FETCH statement.

-201 ** OUT OF MEMORY
Cause: Not enough memory.

-202 ** FILE NOT FOUND
Cause: File not found during SQL_INIT(MULTIPLE%).

-203 ** INVALID FILE
Cause: Invalid file during SQL_INIT(MULTIPLE%).

-204 ** INIT ERROR
Cause: Initialization error during SQL_INIT(MULTIPLE%).

-205 ** SQL_INIT WAS ALREADY USED
Cause: SQL_INIT has been called more than once with different DLLs.

-206 ** LIBRARY ALREADY LOADED
Cause: Library already loaded during SQL_INIT(MULTIPLE%).

-207 ** TOO MANY LIBRARIES OPENED
Cause: The maximum number of libraries that can be opened simultaneously has been reached.

-208 ** CAN'T USE SQL_INITMULTIPLE%, USE SQL_INIT
Cause: SQL_INITMULTIPLE% has been used in single DBMS mode.

-209 ** CAN'T USE SQL_INIT, USE SQL_INITMULTIPLE%
Cause: SQL_INIT has been used in multiple DBMS mode.

-210 ** USE SQL_STOP BEFORE ANOTHER SQL_INIT
Cause: SQL_INIT has been called more than once with different DLLs.

-211 ** INVALID HANDLE
Cause: An invalid handle has been used.

-212 ** LIBRARY NOT LOADED
Cause: An SQL_... function has been used before SQL_INIT(MULTIPLE%).

-213 ** STOP_DATABASE ERROR. LIBRARY MAY NOT BE UNLOADED
Cause: SQL_STOP has been called before SQL_INIT.

-214 ** PATH NOT FOUND
Cause: Library search path not found (during an INIT).

-215 ** TOO MANY OPENED FILES
Cause: Too many files opened simultaneously (during SQL_LOGGINGON).

-216 ** CAN'T ACCESS FILE

Cause: Unable to access file (e.g. attempt to write a log to a protected file).

-217 ** INVALID FILE NAME
Cause: Invalid file name specified.

-218 ** NOT A DOS DISK
Cause: Attempt to access a non-DOS disk.

-219 ** GENERAL OPEN FAILURE
Cause: Unable to open file.

-220 ** DISK FULL
Cause: Disk full while attempting to write to log file.

-221 ** DRIVE IS LOCKED
Cause: Disk is write-protected.

-222 ** SHARING VIOLATION
Cause: Attempt to access a file concurrently.

-223 ** SHARING BUFFER EXCEEDED
Cause: Buffer overflow.

-224 ** WARNING: PROBLEM DURING MODULE LIBERATION
Cause: Cannot free module (during SQL_STOP).

-225 ** INVALID PARAMETER
Cause: Invalid parameter specified.

-226 ** ALREADY LOGGING"
Cause : The trace mode is already active.

-227 ** PARAMETER SIZE GREATER THAN 65535, NOT SUPPORTED IN THIS VERSION"

Cause : The specified size of a host variable is too large.

Summary of supported functions by DBMS

Key:	A = ALL	(applies to all drivers)
	S = SPECIFIC	(specific to driver)
	underlined	(default mode)

	Oracle	**RDB**	**Sybase Système**	**ODBC**	**DB2**	**Microsoft SQL Server**
ANSIOFF / ANSION	A	A	A	A	A	A
IMAGEOFF / IMAGEON	A	A	A	A	A	A
TRIMCHAROFF / TRIMCHARON	A	A	A	A	A	A
GIVECOM INTO :segment_handle	A	A	A	A	A	A
ROWCOUNT INTO :nb_record	A	A	A	A	A	A
STATEMENT INTO :requete_sql	A	A	A	A	A	A
CHANGEDBCNTX :logicaldbname	A	A	A	A	A	A
CALLBACK :window_handle , :user_event	AS	A S	AS	AS	AS	AS
ERRORCOUNT INTO :nb_error	A	A	A	A	A	A
GETERROR :index_error INTO :no_error	A	A	A	A	A	A
KILLQUERY	S		S			S
SETBUFFERSIZE :buffer_size	S		S			S
DESCRIBEOFF / DESCRIBEON	S					
GETDBNAME INTO :logicaldbname	S					
SETCURSORTYPE :cursortype		S				
ASYNCOFF / ASYNCON						S

	Oracle	**RDB**	**Sybase Système**	**ODBC**	**DB2**	**Microsoft SQL Server**
DATAREADY INTO :dataReady						S
QUOTEOFF / QUOTEON			S	S	S	S
GETCURRENTDBCNTX INTO :logicaldbname			S	S	S	S
GETDBNAME INTO :physicaldbname			S	S	S	S
CHARTOHEXAOFF / CHARTOHEXAON			S	S	S	S
SETCURSORMODE :mode			S	S	S	S
CHANGEOPTION :parametre , :option			S	S	S	S
GETTABLE :typobject, :ownername				S	S	
GETTABLEINFO :objecttype,:ownername,:tablename				S	S	
GETCOLUMN :objectname,:ownername,:refname				S	S	
GETINDEXCOLUMN :objectname,:ownername				S	S	
GETPRIMARYKEY :objectname,:ownername				S	S	
GETPROCEDURE				S	S	
GETPROCEDURECOLUMN :objectname,:ownername,:refname				S	S	
GETTYPEINFO :typsql%				S	S	
AUTOCOMMITOFF / AUTOCOMMITON				S	S	
RPCRETCODEOFF / RPCRETCODEON				S	S	
GETINFO :optionname,:status				S	S	

NS_FUNCTION IMAGEOFF, IMAGEON

IMAGEON mode enables binary object management (for example bitmaps.)

IMAGEOFF mode inactivates this function.

Syntax **NS_FUNCTION IMAGEOFF**

and

NS_FUNCTION IMAGEON

Notes

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

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

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

Example

```
; Note: This example applies to Oracle only, because each DBMS
; has its own internal error structure. Each chapter
; has an example for the DBMS it describes.

; ----------------------
; Window INIT event
; ----------------------
GLOBAL HBMP%

; ---------------------------
; Window TERMINATE event
; ---------------------------
DELETEBMP(HBMP%)

; ---------------------------
; Other Window event
; ---------------------------
; -------------------------------------------------------------
; Read bitmap file,
; Insert image in database,
; then retrieve image from database
; -------------------------------------------------------------
LOCAL DEST$(80),DATA%,SIZE%(4),NBREAD%(2),FILE%,NIL%,FNAME$
LOCAL SQL_IMAGE LOCALIMAGE
```

```
; ---- Here the default mode is IMAGEOFF

SQL_EXEC CREATE TABLE T_IMAGE(NUMERO      NUMBER(8),\
                              DESCRIPTION VARCHAR2(80),\
                              IMAGE       LONG RAW)

; ---- Change mode
SQL_EXEC NS_FUNCTION IMAGEON

; ---- Read file and transfer to DATA%
FNAME$ = "C:\WINDOWS\MARQUISE.BMP"
SIZE%=FGETSIZE%(FNAME$)    ; = 25000 in this example
NEW SIZE%,DATA%,"
FILE%=F_OPEN%(1,FNAME$)
F_BLOCKREAD FILE%, DATA%, SIZE%, NBREAD%
IF F_ERROR%
   MESSAGE"ERROR", "Failed to load " & FNAME$ &"!"
   F_CLOSE FILE%
   DISPOSE DATA%
   RETURN 1
ENDIF

; ---- Insert in t_image table
LOCALIMAGE.REALSIZE% = SIZE%
LOCALIMAGE.LENGTH%   = SIZE%
LOCALIMAGE.PTR%      = DATA%
SQL_EXEC INSERT TO T_IMAGE\
         VALUES (1,"An island", :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%

; ---- Retrieval of bitmap from the database
; We have to allocate the maximum amount available because
; we cannot know in advance the size of the image to be selected.
LOCALIMAGE.REALSIZE% = 30000
NEW LOCALIMAGE.REALSIZE%,LOCALIMAGE.PTR%,"
SQL_EXEC SELECT IMAGE INTO:LOCALIMAGE\
         FROM T_IMAGE\
         WHERE NUMERO = 1
IF SQL_ERROR% <> 0
   MESSAGE "SELECT IMAGE",SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
ELSE
   ; ---- Display the image  in the CTRLBMP control
   ;       (here the LOCALIMAGE.length% is 25K)
   FNAME$="C:\WINDOWS\SOUVENIR.BMP"
   FILE%=F_CREATE%(1,FNAME$)
   F_BLOCKWRITE FILE%,\
                LOCALIMAGE.PTR%,\
                LOCALIMAGE.REALSIZE,\
                LOCALIMAGE.LENGTH%
   IF F_ERROR%
      MESSAGE"ERROR", "Failed to write " & FNAME$ &"!"
      F_CLOSE FILE%
      DISPOSE LOCALIMAGE.PTR%
      RETURN 1
   ENDIF
   HBMP%=CREATEBMP%(FNAME$)
   CRTL = HBMP%
   F_CLOSE_FILE%
   DISPOSE LOCALIMAGE.PTR%
ENDIF
DISPOSE LOCALIMAGE.PTR%

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

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

Types for blobs TYPE_SQL_INSERT_BLOB%, TYPE_SQL_SELECT_BLOB%

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

Notes

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

   TYPE_SQL_INSERT_BLOB%

   TYPE_SQL_SELECT_BLOB%

2. They are used for :
   - inserting a binary file into the database
   - retrieving a binary file from the database

Example

```
; Note: This example applies to Oracle only, because each DBMS
; has its own internal error structure. Each chapter
; has an example for the DBMS it describes.
LOCAL NCLVAR HL[4]
LOCAL INT IMAGNO
LOCAL DESCRIP$
LOCAL FIMAGE$
LOCAL INT J
LOCAL SQL$
LOCAL BMP%

SQL_EXEC CREATE TABLE BIGIMAGE(NUMBER      NUMBER(8),\
                               DESCRIPTION VARCHAR2(80),\
                               IMAGE       LONG RAW)


; ---- Insert the contents of BIGFILE.BMP into the BIGIMAGE table
FILL @HL, SIZEOF HL, 0
FIMAGE$         = "C:\WINDOWS\BIGFILE.BMP"
HL[0].PTR_VAR   = @FIMAGE$
HL[0].TYPE_VAR = TYPE_SQL_INSERT_BLOB%
HL[0].SIZE_VAR = SIZEOF FIMAGE$
SQL$="INSERT INTO BIGIMAGE\
             VALUES(1,'This image is larger than > 32K',:)"
SQL_EXEC_LONGSTR @SQL$, @HL, -1

; ---- Select with automatic entry in EXTRACT.BMP
FILL @HL, SIZEOF HL, 0
HL[0].PTR_VAR   = @IMAGNO
HL[0].TYPE_VAR = TYPE_SQL_INT%
HL[0].SIZE_VAR = SIZEOF IMAGNO
HL[1].PTR_VAR   = @DESCRIP$
HL[1].TYPE_VAR = TYPE_SQL_CSTRING%
HL[1].SIZE_VAR = SIZEOF DESCRIP$
FIMAGE$         = "C:\WINDOWS\EXTRACT.BMP"
```

```
HL[2].PTR_VAR  = @FIMAGE$
HL[2].TYPE_VAR = TYPE_SQL_SELECT_BLOB%
HL[2].SIZE_VAR = SIZEOF FIMAGE$

SQL$="SELECT IMAGNO, IMAGFICH, IMAGBUF INTO:,:,: FROM BIGIMAGE"
SQL_EXEC_LONGSTR @SQL$, @HL, -1

; ---- Display image in CTRLBMP control
BMP% = CREATEBMP%(FIMAGE$)
MOVE BMP% TO CONTROLBITMAP
```

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

NS_FUNCTION ANSIOFF, ANSION

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

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

Syntax **NS_FUNCTION ANSIOFF**

and

NS_FUNCTION ANSION

Notes

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

Example

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

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

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

See also SQL_ERROR%, SQL_ERRMSG$

NS_FUNCTION 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

Note

1. TRIMCHAROFF is the default mode.

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$

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.

Variables	*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).

Variables	*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.

Variables	*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.

Variables	*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)
```

See also SQL_EXEC_LONGSTR

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*

Parameter	*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%
```

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

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

Note

1. 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

RECORD, REEXECUTE commands

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

Syntax **RECORD** *SQL-statement*

and

REEXECUTE

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

Notes

1. The parameters in the SQL sequence must still be accessible when the command 'SQL_EXEC REEXECUTE' is issued.

2. After RECORD, any other SQL statement other than REEXECUTE cancels the current RECORD operation.

3. Using RECORD and REEXECUTE allows you to combine the adaptability of dynamic SQL with the speed of static SQL. In fact, a dynamic SQL order is used when the RECORD command is executed. When the REEXECUTE command is executed, as the analysis of query has already been done by the motor, only the values of the host variables are set.

Example

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

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

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

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