

NatStar

Version 6.00 Edition 1

NS-DK

Version 6.00 Edition 1

NatWeb

Version 6.00 Edition 1

Oracle

Information in this document is subject to change without notice as a result of changes in the product. The software described in this document is furnished under a license agreement. The software may be used or copied only in accordance with the terms of the agreement. It is illegal to reproduce the software on any medium unless specifically authorized within the terms of the agreement. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose other than the purchaser's personal use without the written permission of Nat System.

© 2009 Nat System, All rights reserved

Contents

About this Manual	iii
Supported configurations.....	iii
Development environment	iii
Client environment	iii
Server environment	iii
Relationship to other manuals	iv
Organization of the manual	v
Conventions	vi

Chapter 1 Oracle interface

1-1

Introduction	1-4
Correspondence between drivers and the versions of Oracle	1-5
Limitations of instant client	1-6
The SQL*Plus tool	1-6
The ORACLE_HOME variable	1-6
The Windows Registry	1-7
The Tnsnames.ora file	1-8
The NLS_ environment variables.....	1-8
Installation	1-9
Implicit Output Data Conversions	1-10
Functional categories in the NSnnORxx library	1-12
Initializing and stopping application use of the DBMS.....	1-12
Opening and closing a database.....	1-12
Managing the current database	1-12
Choose the DBMS and the base for an SQL command.....	1-12
Executing an SQL command: SELECT, INSERT, UPDATE, CREATE TABLE ...	1-12
Managing the buffers.....	1-12
Managing the cursor	1-13
Running a stored procedure.....	1-13
Recording and reexecuting an SQL command	1-13
Managing blobs and clobs	1-13
Configuring DBMS behavior	1-13
SQL query advanced management	1-14
Handling errors.....	1-14
NSnnORxx library reference	1-15
NS_FUNCTION extensions	1-57
Centralized management of the errors	1-87
Array Processing.....	1-103
Calling Stored Procedures and Stored Functions.....	1-109
Reminder	1-109
Steps Required.....	1-110
Restrictions	1-110
Examples	1-111
Example 1 : simple parameters.....	1-111
Example 2 : simple parameters.....	1-112
Example 3 : Array type parameters	1-113
Example 4 : INPUT Array Parameters	1-114
Example 5 : OUTPUT Array Parameters	1-117
Example 6 : Calling a Stored Function.....	1-120
Example 7 : Calling in DescribeOn Mode.....	1-120
XML Type	1-122

About this Manual

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

Supported configurations

Development environment

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

Client environment

Operating system	DBMS drivers available
Windows NT4, 98, 95 32 bits	Oracle 8.1.7, 8.1.7 XA Oracle 9.0.1, 9.0.1 XA Oracle 9.2.0, 9.2.0 XA
Windows 2000 32 bits	Oracle 8.1.7, 8.1.7 XA Oracle 9.0.1, 9.0.1 XA Oracle 9.2.0, 9.2.0 XA Oracle 10.1, 10.1 XA
Windows XP 32 bits	Oracle 9.0.1, 9.0.1 XA Oracle 9.2.0, 9.2.0 XA Oracle 10.1, 10.1 XA

Server environment

Operating system	DBMS drivers available
Windows NT 4.0 32 bits	Oracle 8.0, 8.0XA Oracle 8.1, 8.1XA Oracle 8.1.7, 8.1.7 XA Oracle 9.0.1, 9.0.1 XA Oracle 9.2.0, 9.2.0 XA
Windows 2000 32 bits	Oracle 8.0, 8.0XA Oracle 8.1, 8.1XA Oracle 8.1.7, 8.1.7 XA Oracle 9.0.1, 9.0.1 XA Oracle 9.2.0, 9.2.0 XA Oracle 10.1, 10.1 XA
Windows 2003 32 bits	Oracle 9.0.1, 9.0.1 XA Oracle 9.2.0, 9.2.0 XA Oracle 10.1, 10.1 XA
AIX 4.1 32bits	Oracle 8.0, 8.0 XA

Operating system	DBMS drivers available
AIX 4.3 32bits	Oracle 8.0, 8.0 XA Oracle 8.1, 8.1 XA Oracle 8.1.7, 8.1.7 XA Oracle 9.0.1, 9.0.1 XA Oracle 9.2.0, 9.2.0 XA
AIX 5.2 32 bits	Oracle 9.2.0 Oracle 10.1, 10.1 XA
HP-UX 10.x 32 bits	Oracle 8.0, 8.0 XA
HP-UX 11.x (Risc 2) 32 bits	Oracle 8.0, 8.0 XA Oracle 8.1, 8.1 XA Oracle 8.1.7, 8.1.7 XA Oracle 9.0.1, 9.0.1 XA Oracle 9.2.0, 9.2.0 XA Oracle 10.1, 10.1 XA
Sun Solaris 2.5 32 bits	Oracle 8.0, 8.0 XA
Sun Solaris 2.7 32 bits	Oracle 8.0, 8.0 XA Oracle 8.1, 8.1 XA
Sun Solaris 9 32 bits	Oracle 8.1.7, 8.1.7 XA Oracle 9.0.1, 9.0.1 XA Oracle 9.2.0, 9.2.0 XA Oracle 10.1, 10.1 XA
Linux RedHat 6.2	Oracle 8.1, 8.1 XA
Linux RedHat Enterprise 3	Oracle 8.1.7, 8.1.7 XA Oracle 9.0.1, 9.0.1 XA Oracle 9.2.0, 9.2.0 XA Oracle 10.1, 10.1 XA

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 useful if you want to program your applications' data accesses yourself.

Organization of the manual

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

Chapter 1

Oracle interface

Describes the functions of the NSnnORxx library associated with the Oracle database.

Conventions

Typographic conventions

Important term	Important terms are printed in bold .
<i>Interface component</i>	The names of windows, dialog boxes, controls, buttons, menus and options are printed in <i>italics</i> .
[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 <i>XXX \ YYY</i>	This means you need to open the <i>XXX</i> menu, then choose the <i>YYY</i> command (option) from this menu. You can perform this action using the mouse or mnemonic characters on the keyboard.
Click the <i>XXX \ YYY</i> button	This means you need to display the tool bar named <i>XXX</i> , then click the <i>YYY</i> 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 <i>XXX</i> button	This means you need to choose the <i>XXX</i> 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)



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



Indicates specific information on using the software under
OS/2-PM 2.x or DOS-Windows 32 bits



Indicates specific information on using the software under
Unix systems

Chapter 1

Oracle interface



The NSnnORxx library enables applications to interact transparently with the services provided by the Oracle database manager. You can access this library via NCL using a number of instructions and functions prefixed by SQL_.



Nat System development tools x.00 support, Oracle versions 8.0 (XA), 8.1(XA), 9.0 (XA), 9.2 (XA), 10g (XA), instant client and 11.



To simplify documentation, NSnnOR8, NSnnOR81, NSnnOR92, NSnnOR101 and NSnnOR11 will be grouped under the generic name of NSnnORxx.

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.
- Oracle specific features

Contents

Introduction	1-4
Correspondence between drivers and the versions of Oracle	1-5
Limitations of instant client	1-6
The SQL*Plus tool	1-6
The ORACLE_HOME variable	1-6
The Windows Registry	1-7
The Tnsnames.ora file	1-8
The NLS_ environment variables	1-8
Installation	1-9
Implicit Output Data Conversions	1-10
➤ <i>Implicit conversions from NCL to Oracle for input variables</i>	
➤ <i>Implicit conversions from Oracle to NCL for output variables</i>	
Functional categories in the NSnnORxx library	1-12
Initializing and stopping application use of the DBMS	1-12
Opening and closing a database	1-12
Managing the current database	1-12
Choose the DBMS and the base for an SQL command	1-12
Executing an SQL command: SELECT, INSERT, UPDATE, CREATE TABLE ...	1-12
Managing the buffers	1-12
Managing the cursor	1-13
Running a stored procedure	1-13
Recording and reexecuting an SQL command	1-13
Managing blobs and clobs	1-13
Configuring DBMS behavior	1-13
SQL query advanced management	1-14
Handling errors	1-14
NSnnORxx library reference	1-15
NS_FUNCTION extensions	1-57
Centralized management of the errors	1-87
Array Processing	1-103
Calling Stored Procedures and Stored Functions	1-109
Reminder	1-109
➤ <i>Syntax for Calling Stored Procedure</i>	
➤ <i>Syntax for Calling Stored Function</i>	
➤ <i>SQL_PROC command parameters, description and values:</i>	
Steps Required	1-110

Restrictions 1-110

Examples 1-111

Example 1 : simple parameters 1-111

Example 2 : simple parameters 1-112

Example 3 : Array type parameters 1-113

Example 4 : INPUT Array Parameters 1-114

➤ *INIT event for the main window*

➤ *TERMINATE event for the main window*

➤ *EXECUTED event for the IN_IN PushButton*

Example 5 : OUTPUT Array Parameters 1-117

➤ *INIT event for the main window*

➤ *TERMINATE event for the main window*

➤ *EXECUTED event for the IN_OUT PushButton*

Example 6 : Calling a Stored Function 1-120

Example 7 : Calling in DescribeOn Mode 1-120

XML Type 1-122

Introduction

The NSnnORxx libraries enable applications built with Nat System development tools to interact transparently with the client version of Oracle.

The libraries use the OCIs of Oracle and are used in deferred mode. The *parse*, *bind* and *execution* of an SQL query require now one single exchange between the client and server.

You can create and call packages, procedures and functions with simple or array type parameters.

The libraries support Oracle's Array Processing mechanisms. Therefore, queries which involve several rows of code can be executed in a single query when calling stored procedures or the following commands: SELECT (FETCH), INSERT, UPDATE, and DELETE.

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

Correspondence between drivers and the versions of Oracle

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 Oracle and the corresponding drivers.

Oracle's version	Driver
Oracle 8.0 (XA)	NSnnOR8.dll
Oracle 8.1 (XA)	NSnnOR81.dll
Oracle 9.0x (XA)	NSnnOR9.dll
Oracle 9.2.x (XA)	NSnnOR92.dll
Oracle 10g (10.1.x) (XA)	NSnnOR101.dll
Oracle 10gR2	NSw2OR102.dll
Oracle instant client 10g	NSnnOR101.dll
Oracle 11	NSnnOR11.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



XA mode implies the use of Tuxedo middleware. For further information, please refer to your documentation « *Client/Server. Development guide* ».



To simplify documentation, NSnnOR73, NSnnOR8, NSnnOR81, NSnnOR92 and NSnnOR101 will be grouped under the generic name of NSnnORxx.



Previous versions of the drivers no longer supported by Nat System in the Windows 32 bit environment are installed in the CONTRIB(S) folder: NSw2OR73.dll, NSw2OR72.dll and NSw2OR71.dll.

Limitations of instant client

Oracle instant client is available on all platforms supported by Oracle for the version 10g. Its interoperability with the server versions is the same one as for the normal client.



For Nat System tools, download on Oracle site, the version of instant client for OCI at the following address:
<http://www.oracle.com/technology/software/tech/oci/instantclient/index.html>

The SQL*Plus tool

The SQL*Plus tool is not integrated in the package of the instant client. If you wish to use it, download the package on the Oracle site for the wished platform and decompress it.

The ORACLE_HOME variable

Oracle instant client can work without positioning the ORACLE_HOME environment variable. For that purpose, indicate the path of its accessible runtime in the current PATH. Besides, it must be positioned in front of other path containing another Oracle runtime.

If this variable is positioned, Oracle instant client recognizes it and uses it by default to search paths as for a normal client.

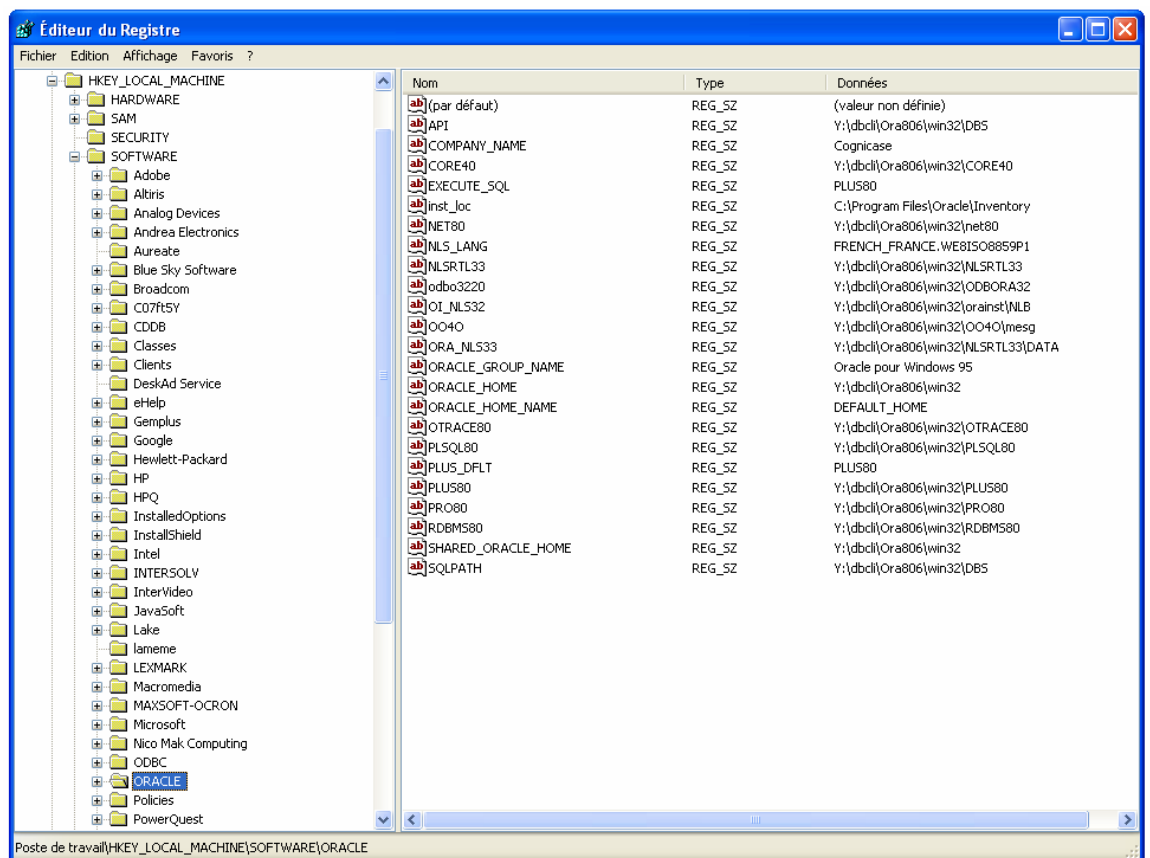


If a normal client is already installed, don't copy the runtime of instant client in \$ORACLE_HOME/lib (Unix) or %ORACLE_HOME%\bin (Windows) directories. Indeed, it could put the normal customer installed in an unstable state. It is thus necessary to copy the runtime of instant client in a different directory and to position the good PATH (or LD_LIBRARY_PATH) to activate according to the need the instant client or the normal client.

The Windows Registry

The relative information at the instant client is not integrated by default into the Windows Registry. But we can use the Registry to set up the instant client.

1. Activate the *Start/Execute* menu. In the *Execute* window, enter `regedit`. The *Register Editor* window appears:



2. Add ORACLE group key under HKEY_LOCAL_MACHINE\SOFTWARE.
3. Add a new string value within the ORACLE key for the different settings you are interested in (NLS_LANG, TNS_ADMIN ...). For keys, that are directories, use a fully qualified path.

The Tnsnames.ora file

For a normal client, the Tnsnames.ora file contains the descriptions of all connection's aliases (scott/tiger@tnsservice.world). It is located in the directory \$ORACLE_HOME/network/admin (Unix) or %ORACLE_HOME%\network\admin (Windows).

To use this file in the case of instant client it is enough to indicate the complete path by the TNS_ADMIN environment variable (or in the Register Windows as indicated above).

The NLS_ environment variables

The ORA_NLS33, ORA_NLS32 and ORA_NLS environment variables are not taken into account with the instant client.

On the other hand, ORA_TZFILE variable is taken into account, but it must indicate the broadest time zone (timezlg.dat for example).

For all other variables (NLS_LANG, NLS_DATE_FORMAT, TNS_ADMIN...) the behavior is the same one as a normal client.

Installation

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

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

Implicit Output Data Conversions

➤ *Implicit conversions from NCL to Oracle for input variables*

To facilitate data insertions and updates in Oracle databases, the NSnnORxx driver transparently carries out implicit conversions of NCL (host variable) to the Oracle motor.

NCL	Oracle
CSTRING, STRING, CHAR	VARCHAR2
CSTRING, STRING, CHAR	VARCHAR
CSTRING, STRING, CHAR	CHAR
CHAR, STRING, CSTRING	CLOB***
INT, NUM	NUMBER
CSTRING, STRING, CHAR	LONG
CSTRING, STRING, CHAR	ROWID
CSTRING, STRING, CHAR	DATE
BLOBS, SEGMENT	RAW
BLOBS, SEGMENT	LONG RAW
CSTRING, STRING,	MLSLABEL

(***) The CLOB type (*Character Large Object*) is available only from Oracle 8.x.

➤ Implicit conversions from Oracle to NCL for output variables

To facilitate data recovery from Oracle databases, the NSnnORxx driver transparently carries out implicit conversions of Oracle data to NCL host variables.

Oracle	NCL
VARCHAR2	CSTRING, STRING, CHAR, INT*, NUM*
VARCHAR	CSTRING, STRING, CHAR, INT*, NUM*
CHAR	CSTRING, STRING, CHAR, INT*, NUM*
CLOB***	CHAR, STRING, CSTRING
NUMBER(p,s)**	INT, NUM, STRING, CSTRING, CHAR
LONG	CSTRING, STRING, CHAR, INT*, NUM*
ROWID	CSTRING, STRING, CHAR
DATE	CSTRING, STRING, CHAR
RAW	BLOBS, SEGMENT
LONG RAW	BLOBS, SEGMENT
MLSLABEL	CSTRING, STRING, CHAR

(*) If the string contains an integer or a real number.

(**) The following rule must absolutely be respected in order not to lose decimal places and therefore precision: if p is greater than or equal to 16, the NCL variable type must be CSTRING or CHAR.

(***) The CLOB type (*Character Large Object*) is available only from Oracle 8.x.

Functional categories in the NSnnORxx library

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

Initializing and stopping application use of the DBMS

SQL_INIT	1-16
SQL_STOP	1-17

Opening and closing a database

SQL_OPEN	1-18
SQL_CLOSE	1-21

Managing the current database

NS_FUNCTION CHANGEDBCNTX	1-62
NS_FUNCTION GETDBNAME	1-70

Choose the DBMS and the base for an SQL command

AT command	1-22
------------------	------

Executing an SQL command: SELECT, INSERT, UPDATE, CREATE TABLE ...

SQL_EXEC	1-23
SQL_EXECSTR	1-27
SQL_EXEC_LONGSTR	1-29

Managing the buffers

NS_FUNCTION SETLONGASTEXT	1-81
NS_FUNCTION SETLONGASBIN	1-83
NS_FUNCTION SETMAXLOBSIZE	1-85

Managing the cursor

SQL_OPENCURSOR%	1-31
SQL_CLOSECURSOR.....	1-33
SQL_OPENTHECURSOR%.....	1-34
SQL_CLOSETHECURSOR.....	1-36

Running a stored procedure

Stored procedure

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

SQL_PROC.....	1-45
---------------	------

Recording and reexecuting an SQL command

RECORD	1-48
REEXECUTE	1-48

Managing blobs and clobs

TYPE_SQL_INSERT_BLOB%	1-50
TYPE_SQL_SELECT_BLOB%.....	1-50
TYPE_SQL_INSERT_CLOB%	1-52
TYPE_SQL_SELECT_CLOB%.....	1-52
TYPE_SQL_CLOB%	1-52

Configuring DBMS behavior

NS_FUNCTION ANSIOFF	1-58
NS_FUNCTION ANSION	1-58
NS_FUNCTION DESCRIBEPUBLICSYNOFF	1-63
NS_FUNCTION DESCRIBEPUBLICSYNON	1-63
NS_FUNCTION DESCRIBEOFF	1-66
NS_FUNCTION DESCRIBEON	1-66
NS_FUNCTION ROLLBACKDBCLOSEOFF.....	1-80
NS_FUNCTION ROLLBACKDBCLOSEON	1-80
NS_FUNCTION IMAGEOFF	1-89

NS_FUNCTION IMAGEON.....	1-89
NS_FUNCTION TRIMCHAROFF	1-99
NS_FUNCTION TRIMCHARON.....	1-99

SQL query advanced management

FOR UPDATE OF	1-54
WHERE CURRENT OF	1-54
NS_FUNCTION GIVECOM.....	1-72
NS_FUNCTION KILLQUERY.....	1-92
NS_FUNCTION ROWCOUNT	1-93
NS_FUNCTION SETBUFFERSIZE	1-94
NS_FUNCTION STATEMENT.....	1-98

Handling errors

SQL_ERROR%.....	1-37
SQL_ERRMSG\$.....	1-44
NS_FUNCTION CALLBACK.....	1-59
NS_FUNCTION ERRORCOUNT	1-69
NS_FUNCTION GETERROR	1-71
NS_FUNCTION AGGREGATEWARNINGOFF	1-88
NS_FUNCTION AGGREGATEWARNINGON	1-88
NS_FUNCTION WARNINGSON	1-101
NS_FUNCTION WARNINGSOFF.....	1-101

NSnnORxx library reference

SQL_INIT instruction

Loads the driver needed to use a given version of Oracle 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 version of Oracle with NCL.
2. The *DLL_name* parameter should contain the name of the DLL used to access the version of Oracle.
3. There is a driver for each target platform and version of Oracle. The format of the *name-DLL* parameter is **NSnnORxx** where:
 - **nn** indicates the target platform: w2 for Windows 32 bits, w4 for Windows 64 bits.
 - **xx** indicates the installed version of Oracle.
4. For the list of available target platform drivers for Oracle versions, please refer to *Correspondence between drivers and the versions of Oracle*.

Example

```
SQL_INIT 'NSw2OR8'      ; loads the DLL working with the client part of
                        ; Oracle 8.0 on a Windows 32 bits environment

SQL_STOP                ; unloads the DLL
```

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

SQL_STOP instruction

Unloads the current DBMS and closes simultaneously all opened databases and cursors.

Syntax **SQL_STOP**

Example
Refer to the 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*, *password-string*

Parameters	<i>logical-DBname</i>	CSTRING	I	logical name of the database to open
	<i>password-string</i>	CSTRING	I	password to connect to a server

Notes

1. Oracle could open several databases through the network.
2. The *logical-DBname* is the name of the database on the Oracle server to open.
3. The *password-string* is a connection string to a local or distant database. Its syntax is :
"USERID[/PASSWORD][@<CONNECTOR>]"

where :

[USERID] name of the user

[PASSWORD] password of the user

[@<CONNECTOR>] connector whose syntax is detailed below

4. The syntax of [@<CONNECTOR>] depends on Sql*Net version:
 - with **Sql*Net V1 (for Oracle 7.1 and Oracle 7.2 only)**, the syntax is :
@<protocol>:<server_name>:<instance_name>

<protocol>: is a letter which corresponds to a communications protocol, that is to say:

X for SPX

T for TCP/IP

P for NMPIPE

B for NETBIOS

With **Sql*Net V1**, if [@<CONNECTOR>] is not specified, the base defined in 'LOCAL' variable of the configuration's file or of the Windows Registry of the customer part employed is used by default.

- With **Sql*Net V2**, CONNECTOR must be equal to the name of a service defined, on the client station, in the file of Oracle TNSNAMES.ORA.
5. The different syntaxes [**@<CONNECTOR>**] for **Oracle instant client 10g** are :
- **@HOST_SERVER:PORT_NUMBER/SID_BASE**
 - **@(DESCRIPTION=**
 (ADDRESS=
 (PROTOCOL=tcp)
 (HOST=host_server)
 (PORT=port_number)
)
 (CONNECT_DATA=
 (SERVICE_NAME=sid_base)
)
)
 - **@TNS_ALIAS**.
 The TNS_ALIAS parameter is defined as an alias of the DESCRIPTION above in the tnsnames.ora file. This file is located in the directory indicated by %TNS_ADMIN% environment variable.
 - **@DNS_SERVER:1521/ORCL**
6. The [**@<CONNECTOR>**] syntax for **Oracle instant client 10g Release 2** is:
@HOST_SERVER:PORT_NUMBER/SID_BASE

Example :

```
Sql_open 'base', 'scott/tiger@service_oracle.world'.
; @HOST_SERVER:PORT_NUMBER/SID_BASE syntax
Sql_open 'base', 'scott/tiger@titanrh:1521/TST10I'
; @ DESCRIPTION ... syntax
Sql_open 'base', 'scott/tiger@(DESCRIPTION = \
    ( ADDRESS = \
        (PROTOCOL = TCP)\
        (HOST = ASEML2)\
        (PORT = 1521)\
    )\
    (CONNECT_DATA = \
        (SERVICE_NAME = utf8)\
    )\
),'
```

7. Refer to the technical handbooks of Oracle to configure the client and server parts Sql*Net V1 or Sql*Net V2. You can also carry out the utilities of Oracle *sqldba*, *sqlplus* or *tnsping* to check the correct operation of your environment before using the NSnnORxx drivers.

8. From now on, the Nat System products version 5.00 support authentication by the network. With this new functionality, it is not necessary any more to specify a login/password to be connected. The authentication is carried out on the database level by the network login directly (domain/user). For that, the administrator of the Oracle database must initially give the right to the various users concerned.



9. Currently, RDBMS connections and the authentication are implemented.

To use an authentication network with Oracle 8.1, add !NETWORK to the connector:

"<USERID>[/<PASSWORD>][@<CONNECTOR>!NETWORK]"

The name of the account user and the password can be empty.

Example

```
; ---- Connection to the logical database BASE2 with
;      Sql*Net V1 and TCP/IP protocol for Oracle 7.1 and Oracle 7.2
;      only
;      th server : SERVER1
;      the instance of database : NAT

SQL_OPEN 'BASE1', 'SCOTT/TIGER@T:SERVER1:NAT'
IF SQL_ERROR% <> 0
    MESSAGE 'Connection's error to BASE1', SQL_ERRMSG$(SQL_ERROR%)
ENDIF

; ---- Connection to the logical database BASE2
;      with Sql*Net V2 through SERVICE_NAT

SQL_OPEN 'BASE2', 'SCOTT/TIGER@SERVICE_NAT'
IF SQL_ERROR% <> 0
    MESSAGE 'Connection's error to BASE1', SQL_ERRMSG$(SQL_ERROR%)
ENDIF
...
SQL_CLOSE 'BASE2'
...
SQL_CLOSE 'BASE1'
```

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

SQL_CLOSE instruction

Closes a connection of a database.

Syntax **SQL_CLOSE** *logical-DBname*

Parameter *logical-DBname* CSTRING I logical name of the database to close

Notes

1. Although that you recommend that you close each of the databases opened by an application, an SQL_CLOSE instruction is automatically generated for each open database when an application is closed.
2. The SQL_CLOSE involves an implicit COMMIT.

Example

Refer to the SQL_OPEN example.

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


AT command

Specifies the name of the logical database affected by the SQL statement that follows. This order allows to carry out an order on a database different from the current one, without changing the current database.

Syntax *AT logical-DBname, SQL-statement*

Parameters	<i>logical-DBname</i>	CSTRING	I	logical database name
	<i>SQL-statement</i>	CSTRING	I	SQL statement to execute

Notes

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

Example

```
SQL_OPEN 'BASE1' , 'SCOTT/TIGER@T:SERVER1:NAT'
SQL_OPEN 'BASE2' , 'SCOTT/TIGER@SERVICE1'
SQL_OPEN 'BASE3' , 'SCOTT/TIGER'

SQL_EXEC SELECT...           ; SELECT on BASE3

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

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

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

SQL_EXEC instruction

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

Syntax **SQL_EXEC** [**AT** *logical-DBname*] *SQL-command* [**USING** *cursor-handle*]

Parameters	<i>logical-DBname</i>	CSTRING	I	logical name of database
	<i>SQL-command</i>	CSTRING	I	SQL command to execute
	<i>cursor-handle</i>	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. For SQL commands that are too long, it is possible to use the special continuation character "\":

```
SQL_EXEC UPDATE SAMPLE SET COMPANY =:A$ \
                                WHERE CITY =:C$ \
AND \
                                COUNTRY =:D$
```
7. The types of variables recognized by the interface are:
 - INT(1), INT(2), et INT(4),
 - NUM(8), NUM(4),
 - STRING,
 - CSTRING,
 - CHAR.

8. 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.
 9. 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] \`
`[, ...]]`
 10. 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.
 11. Always put a ":" before the name of a variable or flag.
 12. 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.
 13. 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 ("").
- 16 bits**
14. The size of the host variables of alpha type (CHAR, etc.) don't have to exceed 255 characters.
 15. To execute automatically an implicit COMMIT command, use the following syntax **SQL_EXEC SET AUTO[COMMIT] { OFF | ON | IMM[EDIATE] | n }**.

Example

```

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

CITY$ = 'PARIS'

; =====
; 1st CASE
; =====
; ---- Selection of sub-unity
SQL_EXEC SELECT CODE, COUNTRY \
          FROM WORLD \
          WHERE CITY = :CITY$
; ---- Reading of the first to the last record of the selection
WHILE SQL_ERROR% = 0
  SQL_EXEC FETCH INTO :CODE%, :COUNTRY$
  IF SQL_ERROR% = 0
    INSERT AT END CODE% && COUNTRY$ TO LBOX1
  ENDIF
ENDWHILE

```

```

; =====
; 2nd CASE (the more performant)
; =====
; ---- Selection of the sub-unity
; and reading of the 1st record of the selection
SQL_EXEC SELECT CODE,COUNTRY \
          FROM MONDE \
          INTO :CODE%,:COUNTRY$ \
          WHERE VILLE = :CITY$
; ---- Reading the 2nd to the last record of the selection
WHILE SQL_ERROR% = 0
  INSERT AT END CODE% && COUNTRY$ TO LBOX1
  SQL_EXEC FETCH
ENDWHILE

; =====
; 3rd CASE
; =====
; ---- Selection of the sub-unity
SQL_EXEC SELECT CODE,COUNTRY \
          FROM WORLD \
          WHERE CITY = :CITY$
; ---- Reading of the 1st to the last record of the selection
; by filling the two arrays TCODE% and TCOUNTRY$
I% = 0
WHILE (SQL_ERROR% = 0) AND (I% < 10)
  SQL_EXEC FETCH INTO :TCODE%[I%],:TCOUNTRY$[I%]
  I% = I% + 1
ENDWHILE

; =====
; Utilisation of the indicators
; =====
SQL_EXEC CREATE TABLE FAMILY(NAME          VARCHAR2(10),\
                              AGE           NUMBER,\
                              NAMECHILD    VARCHAR2(10))

FATHER$ = 'PAUL'
AGE%    = 35
SON$    = 'PIERRE'
IND1%   = 0
IND2%   = 0
; --- Insert in the table ("PAUL",35,"PIERRE")
SQL_EXEC INSERT INTO FAMILY\
          VALUES (:FATHER$:IND1%, :AGE%, :SON$:IND2%)

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

; ---- The loop SELECT displays in the listbox LBOX
; 'The son of PAUL is named PIERRE'
; 'PIERRE has no son'
SQL_EXEC SELECT NAME, AGE, NAMECHILD\
          INTO :FATHER$:IND1%, :AGE%, :SON$:IND2% \

```

```
        FROM FAMILY
WHILE SQL_ERROR% = 0
  ; ---- IND1% is equivalent to 0
  IF IND2% = NULL_VALUE_INDICATOR
    INSERT AT END FATHER$ & ' has no son' TO LBOX
  ELSE
    INSERT AT END 'The son of ' & PERE$ & \
                  'is named' & SON$ TO LBOX
  ENDIF
  SQL_EXEC FETCH
ENDWHILE
```

See also SQL_EXECSTR, SQL_EXEC_LONGSTR, SQL_ERROR%, SQL_ERRMSG\$

SQL_EXECSTR instruction

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

Syntax	SQL_EXECSTR <i>SQL-command</i> [, <i>variable</i> [, <i>variable</i> [,]]] [USING <i>handle-name</i>]			
Parameters	<i>SQL-command</i>	CSTRING	I	SQL order to execute
	<i>variable</i>		I	NCL variable list
	<i>handle-name</i>	INT(4)	I	cursor value

Notes

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

Example

```

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

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

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

; ---- Selection of persons older than 20
;      in the FAMILY table
SQL_EXECSTR REQ$, \
      FATHER$, IND1%, AGE%, SON$, IND2%, AGE% USING CURS1%
WHILE SQL_ERROR% = 0
  IF IND2% = -1
    INSERT AT END FATHER$ & ' has no son' TO LBOX
  ELSE
    INSERT AT END 'The son of ' && FATHER$ && 'is named' && SON$ TO
LBOX
ENDIF

```

```
SQL_EXEC FETCH USING CURS1%  
ENDWHILE  
  
; ---- closing 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	<i>sql-string-address</i>	INT(4)	I	address of the character string containing the SQL statement to execute
	<i>var-array-address</i>	INT(4)	I	address of the array containing the host variables (or indicators)
	<i>cursor-num</i>	INT(2)	I	cursor value

Notes

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

```

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

CONST TYPE_SQL_INT%           0
CONST TYPE_SQL_STRING%       1
CONST TYPE_SQL_CSTRING%      2
CONST TYPE_SQL_NUM%          3
CONST TYPE_SQL_SEGMENT%      10
CONST TYPE_SQL_IMAGE%        11
CONST TYPE_SQL_SELECT_BLOB%   12
CONST TYPE_SQL_INSERT_BLOB%   13
CONST TYPE_SQL_SELECT_XML%    15

```

```

CONST TYPE_SQL_INSERT_XML%      16
CONST TYPE_SQL_CLOB%            14
CONST TYPE_SQL_SELECT_CLOB%     15
CONST TYPE_SQL_INSERT_CLOB%     16

```

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

Example

```

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

; ---- RAZ of the array
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$, @VARLIST, -1
FILL @VARLIST, SIZEOF VARLIST, 0
SQL_STR$ = '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$, @VARLIST, -1
  IF SQL_ERROR% = 0
    MESSAGE 'SELECT', VAR1% && VAR2$
  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%**

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 NSnnORxx 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 NSnnORxx 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

Refer to the example of the SQL_CLOSETHECURSOR instruction.

See also

SQL_CLOSECURSOR, SQL_OPENTHECURSOR%, SQL_CLOSETHECURSOR,
SQL_ERROR%, SQL_ERRMSG\$

SQL_CLOSECURSOR instruction

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

Syntax **SQL_CLOSECURSOR**

Notes

1. SQL_CLOSECURSOR closes the last cursor opened, situated at the top of the LIFO (Last In First Out) cursor stack.
2. SQL_CLOSECURSOR must only close cursors opened with SQL_OPENCURSOR%.
3. The error codes returned by SQL_ERROR% when this instruction is executed could be -32003 ou -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

Refer to the example of the SQL_CLOSETHECURSOR instruction.

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

SQL_OPENTHECURSOR% function

Opens a cursor and returns its handle.

 Prefer this function to SQL_OPENCURSOR%, because it offers a better resources management.

Syntax **SQL_OPENTHECURSOR%**

Return value INT(2)

Notes

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


```
SQL_EXEC SELECT ... USING cursor-handle
SQL_EXEC FETCH ... USING cursor-handle
```
2. A cursor is an internal resource managed by the NSnnORxx 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 a database other than the current one, you must execute the SQL_EXEC CHANGEDBCNTX:otherbase\$ command to change databases before you execute SQL_OPENCURSOR%.

Example



Refer to the example of the SQL_CLOSETHECURSOR instruction.

SQL_CLOSETHECURSOR instruction

Closes the cursor associated with the given handle.

Syntax **SQL_CLOSETHECURSOR** *cursor-handle*

Parameter *cursor-handle* INT(4) I handle of the cursor to close

Note

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

Example

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

SQL_ERROR% function

Returns the error code of the last SQL_ instruction executed.

Syntax **SQL_ERROR%**

Return 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 codes and messages of internal errors:
 - 0 "NSSQLE000 ** UNUSED NATSYS ERROR CODE"
 - 32001 "NSSQLE001 ** HEAP ALLOCATION ERROR"
 - Cause :** Internal memory allocation/deallocation error
 - 32002 "NSSQLE002 ** DYNAMIC ALLOCATION ERROR"
 - Cause :** Internal memory allocation/deallocation error
 - 32003 "NSSQLE003 ** DYNAMIC FREE STORAGE ERROR "
 - Cause :** Internal memory allocation/deallocation error
 - 32004 "NSSQLE004 ** NO MORE CURSORS AVAILABLE"
 - Cause :** Too many cursors opened simultaneously.
 - 32005 "NSSQLE005 ** NO MORE CURSORS OR TRYING TO DEALLOCATE ONLY CURSOR"
 - Cause :** There is no more cursor to free.

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

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

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

Cause : Too many variables in the INTO clause.

-32008 "NSSQLE008 ** MISSING HOST VARIABLE AFTER "

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

-32009 "NSSQLE009 ** INTO CLAUSE : NOT ENOUGH VARIABLES"

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

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

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

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

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

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

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

Cause : Invalid data type in a WHERE clause.

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

Cause : Syntax error in UPDATE WHERE CURRENT OF.

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

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

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

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

-32016 "NSSQLE016 ** INVALID SQL DATA TYPE "

Cause : Data type invalid for output.

-32017 "NSSQLE017 ** INVALID DATA CONVERSION REQUESTED"

Cause : Type conversion invalid for output.

STRING->NUM

NUM->STRING

REAL->INTEGER

INTEGER->REAL

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

- Cause** : Invalid length for the data type (for example, real number with a length of 48)
- 32019 "NSSQLE019 ** INVALID DECIMAL PACKED FORMAT"**
- Cause** : Unable to convert data to packed decimal format.
- +32020 "NSSQLW020 ** STRING DATA TRUNCATED"**
- Cause** : The string passed as a variable is shorter than the field received from the DBMS. The received field has been truncated.
- 32021 "NSSQLE021 ** RESET STORAGE ERROR"**
- Cause** : Error deallocating internal heap.
- +32022 "NSSQLW022 ** FUNCTION NOT SUPPORTED"**
- Cause** : the executed instruction is not available.
- 32023 "NSSQLE023 ** TOO MANY OPENED DATABASES"**
- Cause** : More than 5 databases opened simultaneously.
- 32024 "NSSQLW024 ** DB ALREADY OPENED"**
- Cause** : The database used with SQL_OPEN has already been opened.
- 32025 "NSSQLW025 ** 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)
- 32027 "NSSQLW100 ** NO ROW WAS FOUND OR LAST ROW REACHED"**
- Cause** : Internal error linking with the DBMS.
- 32028 "NSSQLE028 ** UNABLE TO GET Oracle LOGIN "**
- Cause** : Failed to connect to Oracle (e.g. server name error).
- 32029 "NSSQLE029 ** Oracle VARIABLE INPUT BIND FAILED"**
- 32030 "NSSQLE030 ** Oracle VARIABLE OUTPUT BIND FAILED"**
- 32031 "NSSQLE031 ** Oracle BUFFER FILL ERROR"**
- 32032 "NSSQLE032 ** RPC PARAMETER NAME EXPECTED"**
- 32033 "NSSQLE033 ** TOO MANY RPC PARAMETERS"**
- 32034 "NSSQLE034 ** RPC PROCEDURE NAME EXPECTED"**
- 32035 "NSSQLE035 ** NOT ENOUGH PARAMETERS FOR RPC CALLS"**
- 32036 "NSSQLE036 ** INVALID RPC PARAMETERS SUPPLIED"**
- 32037 "NSSQLE037 ** INVALID RPC PROCEDURE INITIALIZATION"**
- 32038 "NSSQLE038 ** RPC PROCEDURE EXECUTION FAILED"**
- 32039 "NSSQLE039 ** MEMORY CONSISTENT ERROR"**
- 32040 "NSSQLE040 ** INVALID TYPE FOR INDICATOR"**
- 32042 "NSSQLE042 ** CONTEXT NOT FOUND"**
- 32047 "NSSQLE047 ** DB NOT AVAILABLE"**
- 32048 "NSSQLE048 ** DB NOT OPENED"**
-

-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"
-32055 "NSSQLE055 ** ERROR IN EXECUTING FETCH COMMAND"
-32056 "NSSQLE056 ** INDICATOR'S SIZE TOO SMALL TO HOLD
VALUE"
-32057 "NSSQLE057 ** INVALID NCL VARIABLE"
-32060 "NSSQLE060 ** RPC: PARAMETER TYPE MISMATCH"
-32061 "NSSQLE061 ** RPC: PROCEDURE NAME MISSING"
-32062 "NSSQLE062 ** RPC: INDICATORS MAY ONLY BE ON OUT
VARIABLES"
-32067 "NSSQLE067 ** LOGGER: CAN'T OPEN FILE"
-32068 "NSSQLE068 ** PARSER: TOKEN TABLE FULL"
-32070 "NSSQLE070 ** ORAEXEC: SQL SERVER ERROR DURING SIZE
BUFFERING EXECUTION"
-32071 "NSSQLE071 ** ORAEXEC: SQL SERVER ERROR DURING
BUFFERING DELETION"
-32072 "NSSQLE072 ** ORAEXEC: INVALID CURSOR MODE"
-32073 "NSSQLE073 ** ORAEXEC: THAT ROW IS NOT IN BUFFER"
-32076 "NSSQLE076 ** ORAEXEC: INVALID SIZE FOR ROW BUFFERING
: USE A SMALLER INTEGER NUMBER"
-32081 "NSSQLE081 ** DATA NOT READY TO RESULT PROCESSING"
-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"
-32090 "NSSQLE090 ** INVALID NS_FUNCTION STATEMENT"
-32091 "NSSQLE091 ** INVALID DATABASE NAME"
-32092 "NSSQLE092 ** STATEMENT BUFFER OVERFLOW : STATEMENT
TOO LONG"
-32095 "NSSQLE095 ** INVALID USER NAME LENGTH"
-32096 "NSSQLE096 ** INVALID PASSWORD LENGTH"
-32097 "NSSQLE097 ** INVALID SERVER NAME LENGTH"

-
- 32099** "NSSQLE099 ** INVALID DATABASE NAME : DBNAME OR USERNAME OR PASSWORD OR SERVER INVALID"
- 32102** "NSSQLE102 ** INVALID CONNECTOR : USERNAME OR PASSWORD OR SERVER INVALID"
- 32103** "NSSQLE103 ** INVALID BUFFER ADDRESS : BUFFER ADDRESS FOR REMOTE PROCEDURE CALL INVALID"
- 32105** "NSSQLE105 ** NOT ENOUGH MEMORY RESSOURCE AVAILABLE : CLOSE UNUSED OPENED CURSORS"
- 32106** "NSSQLE106 ** RETURN PARAMETER FOR A REMOTE FUNCTION CALL IS MISSING; :HOST_VARIABLE = FUNCTION(:HOST_VARIABLE,...)"
- 32107** "NSSQLE107 ** A PARAMETER MODE FOR A REMOTE PROCEDURE CALL IS MISSING SYNTAX IS :HOST_VARIABLE MODE [TYPE, LEVEL]"
- 32108** "NSSQLE108 ** INVALID PARAMETER MODE FOR A REMOTE PROCEDURE CALL; USE IN | OUT | INOUT"
- 32109** "NSSQLE109 ** A PARAMETER TYPE FOR A REMOTE PROCEDURE CALL IS MISSING SYNTAX IS
:HOST_VARIABLE MODE [TYPE, LEVEL]"
- 32110** "NSSQLE110 ** INVALID PARAMETER TYPE FOR A REMOTE PROCEDURE CALL"
- 32111** "NSSQLE111 ** A PARAMETER LEVEL FOR A REMOTE PROCEDURE CALL IS MISSING SYNTAX IS
:HOST_VARIABLE MODE [TYPE, LEVEL]"
- 32112** "NSSQLE112 ** INVALID PARAMETER LEVEL FOR A REMOTE PROCEDURE CALL; USE 0 FOR SCALAR PARAMETER, 1 FOR ARRAY PARAMETER"
- 32113** "NSSQLE113 ** AN OPENED BRACKET FOR A REMOTE PROCEDURE CALL DESCRIPTION IS MISSING SYNTAX IS
:HOST_VARIABLE MODE [TYPE, LEVEL]"
- 32114** "NSSQLE114 ** AN CLOSED BRACKET FOR A REMOTE PROCEDURE CALL DESCRIPTION IS MISSING SYNTAX IS
:HOST_VARIABLE MODE [TYPE, LEVEL]"
- 32115** "NSSQLE115 ** A COLON SEPERATOR BETWEEN TYPE AND LEVEL FOR A REMOTE PROCEDURE CALL DESCRIPTION IS MISSING SYNTAX IS :HOST_VARIABLE MODE [TYPE, LEVEL]"
- 32117** "NSSQLE117 ** BITMAP SIZE IS TOO BIG"
-

-32118 "NSSQLE118 ** INVALID SEQUENCE OF EXECUTION : RECORD MUST BE FOLLOWED BY REEXECUTE"

-32119 "NSSQLE119 ** INVALID NCL TYPE FOR ARRAY"

-32120 "NSSQLE120 ** INVALID NCL TYPE FOR ITEM"

-32121 "NSSQLE121 ** INVALID INTERNAL DATABASE TYPE"

-32122 "NSSQLE122 ** INVALID NCL TYPE FOR SIMPLE PARAMETER IN RPC"

-32123 "NSSQLE123 ** INVALID NCL TYPE FOR ARRAY PARAMETER IN RPC (PASCAL STRING AND SEGMENT TYPE NOT SUPPORTED)"

-32124 "NSSQLE124 ** RECORD TYPE NOT SUPPORTED IN RPC"

-32125 "NSSQLE125 ** ALLOCATION ARRAY TOO SMALL FOR BINDBUFFER"

-32126 "NSSQLE126 ** INVALID LINE NUMBER FOR RPC"

-32127 "NSSQLE127 ** IN OR OUT EXPECTED AFTER BINDBUFFER"

-32128 "NSSQLE128 ** CAN'T OPEN FILE USED FOR IMAGE"

-32129 "NSSQLE129 ** OVERFLOW IMAGE FILE NUMBER"

-32130 "NSSQLE130 ** FILE NAME EXPECTED"

-32131 "NSSQLE131 ** COMPILATION ERROR DURING CREATION PACKAGE, PROCEDURE OR FUNCTION"

-32132 "NSSQLE132 ** (OCI) AN INVALID HANDLE WAS PASSED AS A PARAMETER.",

-32133 "NSSQLE133 ** (OCI) THE APPLICATION MUST PROVIDE RUN-TIME DATA.",

-32134 "NSSQLE134 ** CAN NOT USE INDICATORS IN THIS CASE.",

-32135 "NSSQLE135 ** DO NOT USE MORE THAN ONE CLOB PER REQUEST."

Example

```
...  
  
SQL_EXEC SELECT ENO,ENAME INTO :NO%,:NAME$ FROM EMPLOYE  
WHILE SQL_ERROR% = 0  
    INSERT AT END 'NO=' & NO% & ' NAME=' & NAME$ TO LBOX1  
    SQL_EXEC FETCH INTO :NO%,:NAME$  
ENDWHILE  
IF SQL_ERROR% < 0  
    MESSAGE 'Fatal error', SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)  
ELSE  
    IF SQL_ERROR% > 0  
        MESSAGE 'Warning', SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)  
    ELSE  
        MESSAGE 'OK no error', SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
```

```
ENDIF  
ENDIF  
...
```

See also **Error messages of the chapter *La librairie NSnn_SQL***

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

Return value CSTRING

Notes

1. SQL_ERRMSG\$ returns the last message stored in a work area in the NSnnORxx DLL when the error occurred.
2. This function can be used with all DBMS drivers.
3. Refer to a SQL_ERROR% function for a detailed list of error messages and codes.

Example

See the example of the SQL_ERROR command.

See also **Error messages of the chapter *NSnn_SQL library***

SQL_PROC command

Executes a stored procedure.

Calling procedures syntax

```
SQL_PROC [SCHEMA].PACKAGE_NAME.PROCEDURE_NAME
        {( ... , :host_I {mode [type,level]} , ... , :host_J {mode [type,level]} ,
        ...)
        {VALUES ( ... , buffer_J , ... ) }}
```

By default, SCHEMA is the name of the user's connection which created the stored procedure/function.

Calling functions syntax

```
SQL_PROC :host_returned{mode[type,level]}=
        [SCHEMA].PACKAGE_NAME.FUNCTION_NAME
        {( ... , :host_I {mode [type,level]} , ... , :host_J {mode [type,level]} , ...)
        {VALUES ( ... , buffer_J , ... ) }}
```

By default, SCHEMA is the name of the user's connection which created the stored procedure/function.

Notes

1. SQL_PROC command parameters, description and values:

Parameter	Description and Value
<i>{}</i>	optional
<i>host_returned</i>	Return code of function: NCL variable
<i>host_I</i>	SCALAR parameter: NCL variable
<i>host_J</i>	ARRAY parameter: NCL variable to find out the size and type of element in array
<i>buffer_J</i>	Buffer address containing a single column array associated with the <i>host_J</i> parameter.
<i>mode</i>	IN, OUT, INOUT
<i>type</i>	VARCHAR2, CHAR, CHARZ, STRING, MLSLABEL, VARCHAR, NUMBER, INTEGER, FLOAT, ROWID, DATE, RAW
<i>level</i>	0: for a SCALAR parameter 1: for a ARRAY parameter

1 : pour un parameter de type tableau (array)

2. The *mode*, *type* and *level* parameters exist only in DESCRIBEON mode and explicitly describe the procedure's or function's parameters as they are stored in the database. This avoids requesting a 'describe' for the procedure or function which results in increased performance.

3. Restrictions:

- 16 bits**

 • The maximum size allowed for ALPHA host variables (CHAR, VARCHAR, etc) is 255 characters.
- The size of the host variable must match the size of the parameter passed to the procedure. Therefore, when you declare a string host variable, you must also specify its size: LOCAL A\$(15).

Example

```

;creation of the procedure
SQL_EXEC CREATE OR REPLACE PACKAGE PROC_SIMPLES IS \
PROCEDURE SEL_EMP(DEPART_NO OUT EMP.DEPTNO%TYPE, \
                  EMP_NO      IN OUT EMP.EMPNO%TYPE, \
                  EMP_NAME     OUT EMP.ENAME%TYPE, \
                  EMP_JOB      OUT EMP.JOB%TYPE, \
                  EMP_MGR      OUT EMP.MGR%TYPE, \
                  EMP_SAL      OUT EMP.SAL%TYPE); \
END PROC_SIMPLES;

if sql_error% <> 0
  message 'error PACKAGE' , sql_errmsg$(sql_error%)
endif

SQL_EXEC CREATE OR REPLACE PACKAGE BODY PROC_SIMPLES IS \
PROCEDURE SEL_EMP(DEPART_NO OUT EMP.DEPTNO%TYPE, \
                  EMP_NO      IN OUT EMP.EMPNO%TYPE, \
                  EMP_NAME     OUT EMP.ENAME%TYPE, \
                  EMP_JOB      OUT EMP.JOB%TYPE, \
                  EMP_MGR      OUT EMP.MGR%TYPE, \
                  EMP_SAL      OUT EMP.SAL%TYPE) IS \
BEGIN \
  SELECT EMPNO, ENAME, JOB, MGR, DEPTNO, SAL \
  INTO EMP_NO, EMP_NAME, EMP_JOB, EMP_MGR, DEPART_NO, EMP_SAL \
  FROM EMP \
  WHERE EMPNO = EMP_NO; \
END SEL_EMP; \
END PROC_SIMPLES;

if sql_error% <> 0
  message 'error BODY' , sql_errmsg$(sql_error%)
endif

;call of the procedure

```

```
LOCAL EMP_NO%, DEPART_NO%, MGR_NO%, EMP_SALf
LOCAL CSTRING EMP_NAME(12), CSTRING EMP_JOB(20)

MOVE 7900 TO EMP_NO%

SQL_EXEC SQL_PROC PROC_SIMPLES.SEL_EMP(:DEPART_NO%, :EMP_NO%, \
:EMP_NAME, :EMP_JOB,:MGR_NO%,:EMP_SALf)
if sql_error% <> 0
    message 'error' , sql_errmsg$(sql_error%)
else
    message 'Result', EMP_NO% && EMP_NAME && EMP_JOB && MGR_NO% &&
EMP_SALf
ENDIF
```

See also NS_FUNCTION SETBUFFERSIZE

Examples of the section "Calling stored procedures and stored function".

RECORD, REEXECUTE commands

The RECORD command records an SQL command so that it can be re-executed with REEXECUTE command. When the REEXECUTE command is called, you pass only the new values as parameters.



Syntax **RECORD** *SQL-order*

and

REEXECUTE

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

Notes

1.  The parameters in the sequence SQL must still be valid when SQL_EXEC REEXECUTE is called.
2. After RECORD, any other SQL statement other than REEXECUTE cancels the current RECORD operation.
3. They also support the RECORD REEXECUTE commands which combines 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.
4. These two commands work with all SQL commands with parameters and even with cursors.
5.  The NULL indicators don't work with RECORD and REEXECUTE commands.

Example

```
LOCAL CODE%, i%
LOCAL NAME$(25)
SQL_EXEC DROP TABLE EMP
IF SQL_ERROR% <> 0
    MESSAGE "Warning Drop TABLE", SQL_ERROR%
    &&SQL_ERRMSG$(SQL_ERROR%)
endif
SQL_EXEC CREATE TABLE EMP(EMPNO INTEGER,ENAME CHAR(25))
IF SQL_ERROR% <> 0
    MESSAGE "Warning CREATE TABLE", SQL_ERROR%
    &&SQL_ERRMSG$(SQL_ERROR%)
endif
CODE% = 1
```

```
NAME$ = "NAME1"
SQL_EXEC RECORD INSERT INTO EMP VALUES (:CODE%, :NAME$)
IF SQL_ERROR% <> 0
    MESSAGE "Warning RECORD", SQL_ERROR% &&SQL_ERRMSG$(SQL_ERROR%)
ENDIF

FOR I%= 2 TO 100
    CODE% = I%
    NAME$ = "NAME" & I%
    SQL_EXEC REEXECUTE
    IF SQL_ERROR% <> 0
        MESSAGE "Warning REEXECUTE", SQL_ERROR%
        &&SQL_ERRMSG$(SQL_ERROR%)
    ENDIF
ENDFOR

SQL_EXEC SELECT EMPNO, ENAME FROM EMP INTO :CODE%, :NAME$
WHILE SQL_ERROR% = 0
    INSERT AT END CODE%&&NAME$ TO LISTBOX1
    SQL_EXEC FETCH
ENDWHILE
IF SQL_ERROR% <> 0
    MESSAGE "Warning ", SQL_ERROR% &&SQL_ERRMSG$(SQL_ERROR%)
ENDIF

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

TYPE_SQL_INSERT_BLOB%, TYPE_SQL_SELECT_BLOB% types for blobs

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

Notes

- Two new NCL data types have been added to NSDBMS.NCL and are to be declared in the *Type_Var* field of the NCLVAR structure:
 - TYPE_SQL_INSERT_BLOB%**
 - TYPE_SQL_SELECT_BLOB%**
 They are used for :
 - inserting a binary file into the database
 - retrieving a binary file from the database
- The images inserted with TYPE_SQL_INSERT_BLOB% are not limited in size.
- 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.

Example

```
; WARNING ! This is a specifically example for ORACLE

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)

; ---- Inserting the content of the BIGFILE.BMP file in the BIGIMAGE
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$="INSERT INTO BIGIMAGE\
      VALUES(1,'THIS IS AN IMAGE > 32000 BYTES',:)"
SQL_EXEC_LONGSTR @SQL$, @HL, -1

; ---- Select with automatic writing
;      in the EXTRACT.BMP file
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

; ---- Displaying of the image in the CTRLBMP bitmap
BMP% = CREATEBMP%(FIMAGE$)
MOVE BMP% TO CONTROLBITMAP
```

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

TYPE_SQL_INSERT_CLOB%, TYPE_SQL_SELECT_CLOB%, TYPE_SQL_CLOB% types for CLOB

Enables management of text type objects.

Notes

- 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_CLOB%
 - TYPE_SQL_SELECT_CLOB%
 - TYPE_SQL_CLOB%
 They are used for:
 - insert text a data from a file in a column of the CLOB type.
 - recover a oracle data of the CLOB type in a file.
 - handle directly memories buffers for insertion or selection. Currently, these memory buffers are limited to 64KB. Beyond 64 KB, the use of the files is essential.
- Maximum size of a CLOB for the Oracle 8i and 9i versions is 4GB. For version Oracle 10g, the limiting size is [(4GB-1) * DB_BLOCK_SIZE]. DB_BLOCK_SIZE is a configuration parameter of the base which varies between 2KB and 32KB. Size maximum is thus between 8TB and 128TB (G for Giga, T for Tera, B for Byte).

Example of a file insertion:

```
Local num%
Local NclVar var
Local file$
Local sqlstmt$

SQL_EXEC drop table t_livres
SQL_EXEC create table t_livres (c_num integer not null, c_title
varchar2(255), c_livre clob)

file$ = "d:\livres\code.rtf"
var.type_var = type_sql_insert_clob%
var.ptr_var = @file$
var.size_var = sizeof file$

sqlstmt$ = "insert into t_livres values (1, 'code de la route', :)"
SQL_EXEC_LONGSTR@sqlstmt$, @var, -1
;;; treat the error
```

Example of a selection file:

```
Local num%
Local NclVar var
Local file$
Local sqlstmt$

SQL_EXEC drop table t_livres
SQL_EXEC create table t_livres (c_num integer not null, c_title
varchar2(255), c_livre clob)

file$ = "d:\temp\copie_code.rtf"

var.type_var = type_sql_select_clob%
var.ptr_var = @file$
var.size_var = sizeof file$

sqlstmt$ = "select c_livre from t_livres where c_num = 1 into : "
SQL_EXEC_LONGSTR@sqlstmt$, @var, -1
;;; treat the error
```

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 columns
SELECT ... WHERE ... FOR UPDATE OF List-of-columns
...
... SQL orders of modification
... using columns of List-of-columns
...
; ---- End of transaction and unlock columns
COMMIT ou ROLLBACK
```

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

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

```
; ---- Select and lock the columns
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 ou ROLLBACK
```

Use

1. When using NSnnORxx libraries, the commands are executed dynamically and are based on OCIs. Some SQL commands are only available in *Embedded SQL*, which is the case of the WHERE CURRENT OF clause.
2. When using the NSnnORxx libraries, be sure to respect the following rules:
 - The order in which the selection of 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.

- Oracle's pseudo-column **ROWID** must also be selected with the other columns be added to the list of selected columns selected and the result stored in a NCL CSTRING. This way the address of the record in the active database can be found.
- The order in which the selected record is updated must follow the correct sequence, for example: UPDATE **WHERE ROWID** =:ROWID\$.
- Even though the use of a where-clause in the syntax is required, its execution is immediate taking into account the nature of this pseudo-column.
- When the modification is finished, the transaction must finish with a COMMIT or ROLLBACK, to remove the locks on the *Col-list* columns.

Example

```

LOCAL C1%, C2%, ANC%
LOCAL ROWID$

...
; ---- two cursors must be open
C1% = SQL_OPENTHECURSOR%
C2% = SQL_OPENTHECURSOR%

; ---- Selection with a lock in the BONUS column
SQL_EXEC SELECT ROWID , ANCIENNETE \
          INTO :ROWID$, :ANC% \
          FROM STAFF \
          WHERE SENIORITY >= 2 AND SENIORITY < 25 \
          FOR UPDATE OF BONUS \
          USING C1%
WHILE SQL_ERROR% = 0
  EVALUATE ANC%
  WHERE 2 TO 4 , 6 TO 9
  ; ---- update for the years 2 to 4 and 6 to 9
  SQL_EXEC UPDATE STAFF \
    SET BONUS = BONUS * 1.02 \
    WHERE ROWID = :ROWID$ \
    USING C2%
  ENDWHERE
  WHERE 5,10,15,20
  ; ---- update for the years 5, 10,15 and 20
  SQL_EXEC UPDATE STAFF \
    SET BONUS = BONUS * 1.03 \
    WHERE ROWID = :ROWID$ \
    USING C2%
  ENDWHERE
  ELSE
  ; ---- update for others
  SQL_EXEC UPDATE STAFF \
    SET BONUS = BONUS * 1.01 \
    WHERE ROWID = :ROWID$ \
    USING C2%
  ENDEVALUATE
  ; ---- read the next recording
  SQL_EXEC FETCH USING C1%
ENDWHILE

SQL_CLOSETHECURSOR C1%

```



```
SQL_CLOSETHECURSOR C2%  
; ---- to remove the lock and take account of the updates  
SQL_EXEC COMMIT  
...
```

See also Oracle's manuals for more informations on ROWID pseudo-column.

NS_FUNCTION extensions

The NS_FUNCTIONS correspond to new functionality developed by Nat System to extend database interface options.

The new functionalities can be accessed by the NCL language.

Some commands must be preceded compulsory by the keyword NS_FUNCTION:

SQL_EXEC NS_FUNCTION *command*

NS_FUNCTION ANSIOFF, ANSION

 The two NS_FUNCTION ANSIOFF and ANSION functions have been designed to cope with the following problem. The command SQL_EXEC UPDATE ... WHERE ... sets SQL_ERROR% to 0, even if there is no record.

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

In the ANSION mode, if an UPDATE, DELETE or INSERT 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. ANSION is the default mode.
2. SQL_ERROR% enables you to retrieve the warning returned.

Example

```
; ---- Mode ANSIOFF by default
SQL_EXEC DELETE ... WHERE ...
; ---- Here even if any recordings have been delete
;      SQL_ERROR% equal zéro.

; ---- Mode ANSION
SQL_EXEC NS_FUNCTION ANSION
SQL_EXEC UPDATE ... WHERE ...
IF SQL_ERROR% = 100
    message 'No recording is update',
           SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
ENDIF

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

See also SQL_ERROR%, SQL_ERRMSG\$

NS_FUNCTION CALLBACK


This function enables the redirection of Oracle error messages to an application window. Each time an error message appears, Oracle 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*

Variables	<i>window-handle</i>	INT(4)	I	handle of the window
	<i>user-event</i>	INT(4)	I	user event (USER0 - USER15)

Notes

-  In UNIX, *window-handle* must use the Nat System handle of the window that will receive a notification each time an error occurs.
For all other targets, *window-handle* must be assigned using the NCL GETCLIENTHWNDD%(...) function which receives as input the Nat System handle of the window that will receive a notification each time an error occurs.
- 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.
- To cancel this function, set the window handle to zero.
- 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_ORACLE_CLIENT_STRUCT
  INT    errorType(2)      ; =1 => Warning , =2 => Error
  INT    dbError(2)        ; Error code in abs()
  CHAR   dbErrorStr(512)   ; Error message (with a '\0')
ENDSEGMENT
      
```
- The type of the error's message (clientmsg) is sent to PARAM12%. The handle of the structure is sent to PARAM34%.

Example

```

; Example of activation and deactivation of the management mode of
; centralized errors, with a description example of the NCL code to
; place in the USER1 event in the CATCHERR window.

; -----
; in the INIT event of the MAIN window
; -----
GLOBAL HDLE_CATCHERR%

; -----
; in the TERMINATE event of the MAIN window
; -----
IF HDLE_CATCHERR% <> 0
    CLOSE HDLE_CATCHERR%
    HDLE_CATCHERR%=0
ENDIF

; -----
; in the SELECTED event of the CKCALLBACK checkbox
; (fields input at 0) in the MAIN window
; -----
LOCAL USER_EVENT%
LOCAL WINDOW_HANDLE%

IF CKCALLBACK = CHECKED%
    ; ---- Activate the management mode of centralizd errors
    ; by CALLBACK
    IF HDLE_CATCHERR% = 0
        OPEN_CATCHERR,0,HDLE_CATCHERR%      ; window where
                                              ; the errors will be centralized
    ENDIF
    MOVE GETCLIENTHWND%(HDLE_CATCHERR%) TO WINDOW_HANDLE%
    MOVE 1                                TO USER_EVENT%
    SQL_EXEC NS_FUNCTION CALLBACK :WINDOW_HANDLE% , :USER_EVENT%
    MESSAGE 'START CALLBACK' && WINDOW_HANDLE% && USER_EVENT%",\
        SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
ELSE
    ; ---- Deactivate the management mode of centralized errors
    ; by CALLBACK
    MOVE 0 TO WINDOW_HANDLE%
    MOVE 1 TO USER_EVENT%
    SQL_EXEC NS_FUNCTION CALLBACK :WINDOW_HANDLE% , :USER_EVENT%
    MESSAGE 'STOP CALLBACK' && WINDOW_HANDLE% && USER_EVENT%",\
        SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
    IF HDLE_CATCHERR% <> 0
        CLOSE HDLE_CATCHERR%
        HDLE_CATCHERR%=0
    ENDIF
ENDIF
ENDIF

; -----
; In the USER1 event of the CATCHERR window
; -----
; WARNING ! This example is for Oracle only. Each DBMS has its own
; internal mode of errors management.

```

```

; ---- Recalling definitions contained in NSDBMS.NCL file
;CONST  CLIENTMSG          1
;
;CONST  DB_ORACLE_WARNING 1
;CONST  DB_ORACLE_ERROR   2
;
;SEGMENT DB_ORACLE_CLIENT_STRUCT
;  INT  ERRORTYPE(2)      ; =1 => Warning , =2 => Error
;  INT  DBERROR(2)       ; Error code in absolute value
;  CHAR DBERRORSTR(512) ; Error message (with a '\0')
;ENDSEGMENT

LOCAL MESSAGE% (4)
LOCAL PTR% (4)

MOVE PARAM12% TO MESSAGE%
MOVE PARAM34% TO PTR%


IF MESSAGE% = CLIENTMSG
  EVALUATE DB_ORACLE_CLIENT_STRUCT(PTR%).ERRORTYPE
    WHERE DB_ORACLE_WARNING
      INSERT AT END 'WARNING : +' & \
        DB_ORACLE_CLIENT_STRUCT(PTR%).DBERROR\
        TO LBERR
    ENDWHERE
    WHERE DB_ORACLE_ERROR
      INSERT AT END 'ERROR : -' & \
        DB_ORACLE_CLIENT_STRUCT(PTR%).DBERROR\
        TO LBERR
    ENDWHERE
  ENDEVALUATE
  ; display DB_ORACLE_CLIENT_STRUCT(ptr%).dbErrorStr in LBERR
ELSE
  INSERT AT END 'MESSAGE TYPE UNKNOW' TO LBERR
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 was developed to carry out SQL queries on different databases.



To move from one database to another, we recommend using the command NS_FUNCTION CHANGEDBCNTX. It is likely that the AT command will not be supported in future versions.

Syntax	NS_FUNCTION CHANGEDBCNTX : <i>logical-DBname</i>		
Variable	<i>logical-DBname</i>	CSTRING I	logical name of the database which becomes the current one

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 LOGICALDBNAME$
SQL_OPEN "BASE1", "SCOTT/TIGER@SERVICE1"
SQL_EXEC .... ; BASE1 is the current database

SQL_OPEN "BASE2", "SCOTT/TIGER@SERVICE2"
SQL_EXEC .... ; BASE2 is the current database

LOGICALDBNAME$ = "BASE1"
SQL_EXEC NS_FUNCTION CHANGEDBCNTX :LOGICALDBNAME$
SQL_EXEC .... ; BASE1 is the current database
SQL_CLOSE "BASE1"
SQL_EXEC .... ; BASE2 is the current database

SQL_CLOSE "BASE2"
```

See also SQL_OPEN, SQL_CLOSE, AT, SQL_ERROR%, SQL_ERRMSG\$, SQL_EXEC

NS_FUNCTION DESCRIBEPUBLICSYNOFF, DESCRIBEPUBLICSYNON



These two functions are only necessary if the parameters of the stored procedure/function called are not described explicitly.

From Oracle 8.x, to call a stored function/procedure by its synonym without explicitly describing its parameters, NS_FUNCTION DESCRIBEPUBLICSYNON should be called first.

Syntax

NS_FUNCTION DESCRIBEPUBLICSYNOFF
and
NS_FUNCTION DESCRIBEPUBLICSYNON

Notes

1. NS_FUNCTION DESCRIBEPUBLICSYNOFF is the default mode.
2. DESCRIBEPUBLICSYNON mode enables a description of the engine's parameters to be obtained for the parameters not described.
3. DESCRIBEPUBLICSYNOFF mode enables DESCRIBEPUBLICSYNON mode to be disabled.

Exemple

```
;this is the stored procedure created with a synonym

SQL EXEC CREATE OR REPLACE PACKAGE PROC_SIMPLES IS \
PROCEDURE SEL_EMP(DEPART_NO OUT EMP.DEPTNO%TYPE, \
                  EMP_NO IN OUT EMP.EMPNO%TYPE, \
                  EMP_NAME OUT EMP.ENAME%TYPE, \
                  EMP_JOB OUT EMP.JOB%TYPE, \
                  EMP_MGR OUT EMP.MGR%TYPE, \
                  EMP_SAL OUT EMP.SAL%TYPE); \
END PROC_SIMPLES;
if sql_error% <> 0
    message 'error PACKAGE' , sql_errmsg$(sql_error%)
endif
```



```

SQL_EXEC CREATE OR REPLACE PACKAGE BODY PROC_SIMPLES IS \
  PROCEDURE SEL_EMP(DEPART NO OUT      EMP.DEPTNO%TYPE, \
    EMP_NO      IN OUT EMP.EMPNO%TYPE, \
    EMP_NAME    OUT      EMP.ENAME%TYPE, \
    EMP_JOB     OUT      EMP.JOB%TYPE, \
    EMP_MGR     OUT      EMP.MGR%TYPE, \
    EMP_SAL     OUT      EMP.SAL%TYPE) IS \
  BEGIN \
    SELECT EMPNO,  ENAME,  JOB,      MGR,      DEPTNO,  SAL \
    INTO  EMP_NO, EMP_NAME, EMP_JOB, EMP_MGR, DEPART_NO, EMP_SAL \
    FROM EMP \
    WHERE EMPNO = EMP_NO; \
  END SEL_EMP; \
END PROC_SIMPLES;

if sql_error% <> 0
  message 'error BODY' , sql_errmsg$(sql_error%)
endif
;grant on the package
SQL_EXEC grant all on PROC_SIMPLES to public

if sql_error% <> 0
  message 'error Grant' , sql_errmsg$(sql_error%)
endif

SQL_EXEC DROP PUBLIC synonym SIMPLES
if sql_error% <> 0
  message 'DROP Synonym' , sql_errmsg$(sql_error%)
endif
SQL_EXEC create public synonym SIMPLES for PROC_SIMPLES

if sql_error% <> 0
  message 'error Synonym' , sql_errmsg$(sql_error%)
endif

;we can call it with different manners by describing parameters like
;this :
LOCAL EMP_NO%, DEPART_NO%, MGR_NO%, EMP_SALf
LOCAL CSTRING EMP_NAME(12), CSTRING EMP_JOB(20)

MOVE 7900 TO EMP_NO%

SQL_EXEC SQL_PROC SCOTT.PROC_SIMPLES.SEL_EMP (:DEPART_NO% OUT [NUMBER,
0], \
:EMP_NO% INOUT [NUMBER, 0], :EMP_NAME  OUT [VARCHAR2, 0], :EMP_JOB  OUT
[VARCHAR2, 0], \
:MGR_NO%  OUT [NUMBER, 0], :EMP_SALf OUT [NUMBER, 0])
if sql_error% <> 0
  message 'error' , sql_errmsg$(sql_error%)
else
  message 'Result', EMP_NO% && EMP_NAME && EMP_JOB && MGR_NO% &&
EMP_SALf
ENDIF
; or with the synonym

```

```

LOCAL EMP_NO%, DEPART_NO%, MGR_NO%, EMP_SAL$
LOCAL CSTRING EMP_NAME(12), CSTRING EMP_JOB(20)

MOVE 7900 TO EMP_NO%

SQL EXEC SQL_PROC SIMPLES.SEL_EMP (:DEPART_NO% OUT [NUMBER, 0], \
:EMP_NO% INOUT [NUMBER, 0], :EMP_NAME OUT [VARCHAR2, 0], :EMP_JOB OUT
[VARCHAR2, 0], \
:MGR_NO% OUT [NUMBER, 0], :EMP_SAL$ OUT [NUMBER, 0])
if sql_error% <> 0
    message 'error' , sql_errmsg$(sql_error%)
else
    message 'Result', EMP_NO% && EMP_NAME && EMP_JOB && MGR_NO% &&
EMP_SAL$
ENDIF
;Or without description of parameters. Thus you must use
DescribePublicSynOn if
;the call is made with the synonym
LOCAL EMP_NO%, DEPART_NO%, MGR_NO%, EMP_SAL$
LOCAL CSTRING EMP_NAME(12), CSTRING EMP_JOB(20)

MOVE 7900 TO EMP_NO%
SQL EXEC NS_FUNCTION DescribePublicSynOn
if sql_error% <> 0
    message 'error BODY' , sql_errmsg$(sql_error%)
endif
SQL EXEC SQL_PROC SIMPLES.SEL_EMP(:DEPART_NO%, :EMP_NO%, :EMP_NAME,
:EMP_JOB,:MGR_NO%,:EMP_SAL$)
if sql_error% <> 0
    message 'error' , sql_errmsg$(sql_error%)
else
    message 'Result', EMP_NO% && EMP_NAME && EMP_JOB && MGR_NO% &&
EMP_SAL$
ENDIF
;but if the call is prefixed by the schema, we don't need anymore
;DescribePublicSynOn
LOCAL EMP_NO%, DEPART_NO%, MGR_NO%, EMP_SAL$
LOCAL CSTRING EMP_NAME(12), CSTRING EMP_JOB(20)

MOVE 7900 TO EMP_NO%
SQL EXEC SQL_PROC SCOTT.PROC_SIMPLES.SEL_EMP(:DEPART_NO%, :EMP_NO%,
:EMP_NAME, :EMP_JOB,:MGR_NO%,:EMP_SAL$)
if sql_error% <> 0
    message 'error' , sql_errmsg$(sql_error%)
else
    message 'Result', EMP_NO% && EMP_NAME && EMP_JOB && MGR_NO% &&
EMP_SAL$
ENDIF
;to return to default mode, do this
SQL EXEC NS_FUNCTION DescribePublicSynOff
if sql_error% <> 0
    message 'error BODY' , sql_errmsg$(sql_error%)
endif

```

See also

SQL_PROC

The examples in the « Calling stored procedures and stored function » section.

NS_FUNCTION DESCRIBEON, DESCRIBEON



These two functions are obsolete. Oracle automatically detects the mode (implicit or explicit) being used to call stored procedures or functions. The syntax analyzer in the NsnnORxx library automatically puts itself into DESCRIBEON mode as soon as it detects a hook opening after an SQL_PROC. Otherwise, it puts itself into DESCRIBEOFF mode. If you are already using these two functions in your code, there is no need to remove them, but understand that they have no effect.

Sets the calling mode of stored procedures and functions.

Syntax

```
NS_FUNCTION DESCRIBEON
and
NS_FUNCTION DESCRIBEOFF
```

Notes

1. DESCRIBEOFF is the mode by default.
2. In DESCRIBEOFF mode, the description of parameters is implicit for the user, but leads to a 'describe' request for the procedure or function to the server on which they are found.
3. In DESCRIBEON mode, the description must be requested by the user and identical to those used when creating the procedure or function. This technique improves performance because this avoids requesting a 'describe' for the procedure or function from the server and results in increased system performance.
4. In explicit mode, the user has to specify the *mode*, *type* and *level* parameters for each host variable used. See the SQL_PROC instruction for more information about the possible values of these three values.


```
VALUES (:HTB%)  
;... After the call, the array has been filled by the procedure  
;   the NB% variable has been modified
```

See also SQL_PROC, DESCRIBEPUBLICSYNOFF, DESCRIBEPUBLICSYNON
 The examples in the « Calling stored procedures and stored function » section.

NS_FUNCTION ERRORCOUNT

Retrieves the number of errors or error messages encountered while executing a query.



The NS_FUNCTION ERRORCOUNT is useful principally to keep compatibility with other databases like Microsoft SQL Server or Sybase.

Syntax	NS_FUNCTION ERRORCOUNT INTO : <i>nbr-errors</i>			
Variable	<i>nbr-errors</i>	INT(4)	O	number of errors encountered while executing a query

Notes

1. The messages of error are numbered 0 or 1.
2. Only the last message is accessible.

Example

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

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

NS_FUNCTION GETDBNAME

Retrieves the logical name of the current database.

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

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

Example

```
LOCAL DBNAME$,DBNAME2$

SQL_OPEN "BASE1","SCOTT/TIGER@T:MACHINE:SERV1"

SQL_OPEN "BASE2","SCOTT/TIGER@SERVICE_VENTE"

SQL_EXEC NS_FUNCTION GETDBNAME INTO :DBNAME$
MESSAGE 'THE CURRENT DATABASE IS :',DBNAME2$
; displays the logical name BASE2

SQL_CLOSE "BASE2"
SQL_EXEC NS_FUNCTION GETDBNAME INTO :DBNAME$
MESSAGE 'THE CURRENT DATABASE IS :',DBNAME$
; displays the logical name BASE1
```

See also SQL_OPEN, SQL_CLOSE, NS_FUNCTION CHANGEDBCNTX

NS_FUNCTION GETERROR

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



The NS_FUNCTION GETERROR is used principally to keep the compatibility with other databases like Microsoft SQL Server or Sybase.

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

Variables	<i>error-index%</i>	INT(4)	I	index of the error number
	<i>error-nbr%</i>	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 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*

Variable *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 initd(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_NCLEMENT (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_NCLEMENT
  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_STATEMENTS**

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	<i>COM_BUFFER%</i>	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	<i>COM_BUFFER%</i>	INT(4)	Handle on COM_AREA
	<i>COLUMN%</i>	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	<i>COM_BUFFER%</i>	INT(4)	Handle on COM_AREA
	<i>COLUMN%</i>	INT(2)	Order of the NCL variable

Return value INT(2)

- **FUNCTION SQL_GET_HOSTCOLUMNTYPES**

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

Variable	<i>TYPE%</i>	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	<i>COM_BUFFER%</i>	INT(4)	Handle on COM_AREA
	<i>COLUMN%</i>	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	<i>COM_BUFFER%</i>	INT(4)	Handle on COM_AREA
	<i>COLUMN%</i>	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% , HEADERS$, A$
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
    ; retrieving the string command
    UPDATE LISTBOX1
    IF SQL_GET_STATEMENT%(COM_AREA_RET%) <> 1
        ; the value of the command is different of the SELECT
        RETURN 1
    ENDIF

    MOVE SQL_GET_HOSTPTR%(COM_AREA_RET%) TO NCL_PTR%
    ; retrieving the handle in the NCL variables array
    MOVE SQL_GET_NBCOL%(COM_AREA_RET%) +
SQL_GET_NBCOMPUTE%(COM_AREA_RET%) \
    TO TOTAL_COL%
    ; retrieve the number of COMPUTE type columns
    IF SQL_GET_LENGTHFETCH%(COM_AREA_RET%) <> 0
        ; if the size of FETCH buffer 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%
            ; retrieving an handle containing a NCL variable
            IF BUFFER_PTR% = 0
                MOVE I% + 1 TO I%
                CONTINUE
            ENDIF
            MOVE " " to HEADERS$
            IF SQL_GET_SQLCOLUMNSSERVICE% (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
; retrieving 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$(COM_AREA_RET% , I% ) && ':' && \
ASC%
(COM_INT1(BUFFER_PTR%).i1) TO LISTBOX1
ENDWHERE
WHERE 2 ; it is an integer of 2
; retrieve the name of the column
INSERT AT END HEADER$ && \

SQL_GET_SQLCOLUMNNAME$(COM_AREA_RET% ,i% )&& \
\:' &&
COM_INT2(BUFFER_PTR%).i2 TO LISTBOX1
ENDWHERE
WHERE 4 ; it is an integer of 4
INSERT AT END HEADER$ && \

SQL_GET_SQLCOLUMNNAME$(COM_AREA_RET% ,i% )&& \
\:' &&
COM_INT4(BUFFER_PTR%).i4 TO LISTBOX1
ENDWHERE
ENDEVALUATE
ENDWHERE
WHERE 2 ; is a C string
INSERT AT END HEADER$ && \
SQL_GET_SQLCOLUMNNAME$(COM_AREA_RET% , I% ) && \
\:' && COM_STRING(BUFFER_PTR%).CS TO LISTBOX1
ENDWHERE
WHERE 3 ; it is a real
EVALUATE
SQL_GET_HOSTCOLUMNLENGTH%(COM_AREA_RET% ,\ I%)
; retrieve the size of the WHERE 4 column
; retrieve the name of the column and the
value
; and the value of the real is the size of 4
INSERT AT END HEADER$ && \

SQL_GET_SQLCOLUMNNAME$(COM_AREA_RET% , I% ) && \
\:' && COM_FLOAT4(BUFFER_PTR%).f4 TO LISTBOX1
ENDWHERE
WHERE 8 ;
; retrieve the name and the value of the
column
; of 8 real
INSERT AT END HEADER$ && \

SQL_GET_SQLCOLUMNNAME$(COM_AREA_RET% , I% ) && \
\:' &&
COM_FLOAT8(BUFFER_PTR%).f8 TO LISTBOX1

```

```

                                ENDWHERE
                                ENDEVALUATE
                                ENDWHERE
                                ELSE
                                ; retrieve the NCL type column
                                INSERT AT END 'NCLType' && 'INVALID' && \
                                SQL_GET_HOSTCOLUMNTYPE%(COM_AREA_RET%, i%) TO
\ LISTBOX1
                                ; return the NCL type 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 handle on the FETCH
                                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 ROLLBACKDBCLOSEOFF, ROLLBACKDBCLOSEON

The default mode on closing a connection to the database is the implicit validation of the actions carried out.

The NS_FUNCTION ROLLBACKDBCLOSEON disables automatic validation of the actions carried out on closing a connection to the database.

Syntax **NS_FUNCTION ROLLBACKDBCLOSEOFF**
 and
 NS_FUNCTION ROLLBACKDBCLOSEON

Notes

1. ROLLBACKDBCLOSEOFF is the default mode.
2. In ROLLBACKDBCLOSEOFF mode, where the connection to the database is closed, actions carried out are automatically validated.
3. In ROLLBACKDBCLOSEON mode, where the connection to the database is closed, actions carried out are automatically cancelled.

Example

```
SQL_INIT "NSw2OR81"
SQL_OPEN "Pubs", "Scott/Tiger@pubs"
; in case the application hangs all updates will be rolled back.
SQL_EXEC NS_FUNCTION ROLLBACKDBCLOSEON
...unqualified Ncl Code ....
SQL_EXEC INSERT.....
....
SQL_EXEC DELETE .....
....
SQL_EXEC UPDATE ....
....
...END of unqualified Ncl Code ....

;default mode
SQL_EXEC NS_FUNCTION ROLLBACKDBCLOSEOFF
```

NS_FUNCTION SETLONGASTEXT

The NS_FUNCTION SETLONGASTEXT handles the buffers to be inserted in LONG type columns, as text.



This function has been developed to improve performances.



This function is useful only if the size of the host string is superior to 2000 bytes.

Syntax **NS_FUNCTION SETLONGASTEXT**

Note

1. Instead of send the maximum size of LENGTH (30000), the driver send the real size of the text with two additional bytes at the Oracle database. This optimizes the send of data packages through the network.

Example

```
local MaChaine$ (20)
local var1%

;Comment:
;setlongasbin: ns_function used to consider long type buffer as binary.
;The driver will send the size of the Oracle long type buffer (30000)
to Oracle.
;setlongastext: ns_function used to manipulate the buffer like a
string.
;Instead of sending the whole long buffer, the driver will only send
the length of the string with an additional 2 bytes to the Oracle
database.

;table TEST ( ID NUMBER(5), DATA LONG)

MaChaine$='Hello world'
var1%=1
;To insert long char values as a Text, you must call this ns_function :
;it'll send many bytes less over the network thus enhances the response
time
SQL_EXEC NS_FUNCTION SETLONGASTEXT
IF SQL_ERROR% <> 0
    MESSAGE 'Warning', SQL_ERROR% &&SQL_ERRMSG$(SQL_ERROR%)
endif

sql_exec INSERT INTO TEST (ID, DATA) VALUES (:var1%, :machaine$)
IF SQL_ERROR% <> 0
    MESSAGE 'Error', SQL_ERROR% &&SQL_ERRMSG$(SQL_ERROR%)
endif
```

```
SQL EXEC COMMIT
IF SQL_ERROR% <> 0
    MESSAGE 'Warning', SQL_ERROR% &&SQL_ERRMSG$(SQL_ERROR%)
endif

;To return to the default mode use this ns_function :
SQL_EXEC NS_FUNCTION SETLONGASBIN
IF SQL_ERROR% <> 0
    MESSAGE 'Warning', SQL_ERROR% &&SQL_ERRMSG$(SQL_ERROR%)
Endif
```

See also NS_FUNCTION SETLONGASBIN

NS_FUNCTION SETLONGASBIN

This function allows to return to the standard mode of handling the buffers. This mode consists in inserting buffers in the columns of LONG type, like binary code. In this case, the maximum size LENGTH (30000) is sent to the base.



This function is useful only if the size of the host string is superior than 2000 bytes.

Syntax **NS_FUNCTION SETLONGASBIN**

Example

```
LOCAL  MaChaine$ (20)
local var1%

;Comment:
;setlongasbin: ns_function used to consider long type buffer as binary.
;The driver will send the size of the Oracle long type buffer (30000)
to Oracle.
;setlongastext: ns_function used to manipulate the buffer like a
string.
;Instead of sending the whole long buffer, the driver will only send
;the length of the string with an additional 2 bytes to the Oracle
database.

;table TEST (ID NUMBER(5), DATA LONG)

MaChaine$='Hello world'
var1%=1
;To insert long char values as a Text, you must call this ns function :
;it'll send many bytes less over the network thus enhances the response
time
SQL EXEC NS FUNCTION SETLONGASTEXT
IF SQL_ERROR% <> 0
    MESSAGE 'Warning', SQL_ERROR% &&SQL_ERRMSG$(SQL_ERROR%)
endif

sql_exec INSERT INTO TEST (ID, DATA) VALUES (:var1%, :machaine$)
IF SQL_ERROR% <> 0
    MESSAGE 'Error', SQL_ERROR% &&SQL_ERRMSG$(SQL_ERROR%)
endif

SQL_EXEC COMMIT
IF SQL_ERROR% <> 0
    MESSAGE 'Warning', SQL_ERROR% &&SQL_ERRMSG$(SQL_ERROR%)
endif

;To return to the default mode use this ns_function :
SQL_EXEC NS_FUNCTION SETLONGASBIN
```

```
IF SQL_ERROR% <> 0
    MESSAGE 'Warning', SQL_ERROR% &&SQL_ERRMSG$(SQL_ERROR%)
Endif
```

See also NS_FUNCTION SETLONGASTEXT

NS_FUNCTION SETMAXLOBSIZE

Splits big data up (CLOB, BLOB, XML, ...) in bytes packages. It re-dimensions the transfer buffer during the insertion or selection of big size data in or from an Oracle database (XML files for instance).



This NS-FUNCTION is available from Oracle 9.0.

Syntax	NS_FUNCTION SETMAXLOBSIZE :lobsize%			
Parameter	lobsize%	INT(4)	I	size of the bytes package

Notes

1. By default, the size of the bytes packages is set to 32 ko.
2. There is no maximal limit for the size of the transfer buffer, but the available memory.
3. For example, it permits to insert or retrieve an XML file of 20 Mo without inducing a memory fault error.

Example

```
; retrieve an XML data directly in a file
local num%
local xml$ (3000)
local nclvar hl[2]
local sql$
local lobsize%

num% = ef_num
fill xml$, 0, sizeof xml$

xml$ = 'c:\temp\cleopatre.xml'

hl[0].ptr_var = @xml$
hl[0].type_var = type_sql_select_xml% ;
hl[0].size_var = sizeof xml$

hl[1].ptr_var = @num%
hl[1].type_var = type_sql_int%
hl[1].size_var = 4

;in the case of a big chunk of data and in order to optimise the
retrieval by
;packages, we can increase the size of the transfer buffer of SqlNet.
The default ;size is 32 ko

lobsize% = 500*1024 ; 512k
```

```
sql_exec ns_function setmaxlobsize :lobsize%
message 'setmaxlobsize' && sql_error%, sql_errmsg$ (sql_error%)

;by calling this function, a data of 512k will be retrieve in one go,
otherwise
;it would be retrieved in packages of 32k(default), so 512/32=16 times
(time consuming)

sql$ = 'select t.xml_col.getclobval ()  into :  from tabdemoxml t where
t.int_col = : '

sql_exec_longstr @sql$, @hl, -1
message 'select xml file' && sql_error%, sql_errmsg$ (sql_error%)
```

Centralized management of the errors

This mode of error management permits more precise and powerful error management than that carried out by SQL_ERROR%, SQL_ERRMSG\$, NS_FUNCTION ERRORCOUNT and NS_FUNCTION GETERROR.

NS_FUNCTION AGGREGATEWARNINGOFF, AGGREGATEWARNINGON

NS_FUNCTION AGGREGATEWARNINGOFF allows to have an ascending compatibility between Oracle 7 and 8 by ignoring the 24347 warning only. This warning appears when we call an aggregate function on a table containing null data.

NS_FUNCTION AGGREGATEWARNINGON does not ignore warning 24347.



This NS-FUNCTION is available from Oracle 8.0.

Syntax **NS_FUNCTION AGGREGATEWARNINGOFF**
 and
 NS_FUNCTION AGGREGATEWARNINGON

Note

1. AGGREGATEWARNINGON is the default mode.

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


2. IMAGEOFF is the default mode.

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. 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%).

5. Images are not the only type of binary objects. Any type of binary file can be stored.

6.  Binary storage is not cross-platform. Oracle column receives an image which should be LONG RAW type.



It is not possible to use or create a table with more than one column of LONG RAW or LONG types.

Example

```
;creation of the table
SQL EXEC CREATE TABLE T_IMAGE(NUMBER NUMBER(8), DESCRIPTION
VARCHAR2(80), IMAGE LONG RAW)
if sql_error% <> 0
  message 'error Create' , sql_errmsg$(sql_error%)
endif

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

```

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

SIZE%=FGETSIZE%(FNAME$)

INSERT AT END 'SIZE' & SIZE% TO LISTBOX1
NEW SIZE%,DATA%
FILE%=F_OPEN%(1,FNAME$)
F_BLOCKREAD FILE%, DATA%, SIZE%, NBREAD%
IF F_ERROR%
  MESSAGE 'ERROR', 'Failed to load' & FNAME$ & '!'
  F_CLOSE FILE%
  DISPOSE DATA%
  RETURN 1
ENDIF
; ---- Insert into table t_image
LOCALIMAGE.REALSIZE = SIZE%
LOCALIMAGE.LENGTH% = SIZE%
LOCALIMAGE.PTR% = DATA%
SQL_EXEC INSERT INTO T_IMAGE VALUES (1,'An island between the sky and
the \
water', :LOCALIMAGE)
IF SQL_ERROR% <> 0
  MESSAGE 'INSERT IMAGE', SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
  F_CLOSE FILE%
  DISPOSE DATA%
  RETURN 1
ENDIF
F_CLOSE FILE%
DISPOSE DATA%
; SELECT
LOCAL DEST$(80),DATA%,SIZE%(4),NBREAD%(2),FILE%,NIL%,FNAME$, hbmp%
LOCAL SQL_IMAGE LOCALIMAGE
SQL_EXEC NS_FUNCTION IMAGEON

IF SQL_ERROR% <> 0
  MESSAGE 'IMAGEON', SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
  RETURN 1
ENDIF
LOCALIMAGE.realsize = 30000
NEW LOCALIMAGE.realsize,LOCALIMAGE.PTR%
SQL_EXEC SELECT IMAGE INTO :LOCALIMAGE FROM T_IMAGE WHERE NUMBER = 1
IF SQL_ERROR% <> 0
  MESSAGE 'SELECT IMAGE',SQL_ERROR% && SQL_ERRMSG$(SQL_ERROR%)
ELSE

```

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

See also

NSDBMS.NCL, SQL_ERROR%, SQL_ERRMSG\$,
TYPE_SQL_INSERT_BLOB%, TYPE_SQL_SELECT_BLOB%

NS_FUNCTION 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.



This function is useful for Oracle 7.x. From Oracle 8.x, it is obsolete.

Syntax **NS_FUNCTION KILLQUERY**

Note

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

Example

```
LOCAL I%, PRIXTTC%, TOTAL%

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

NS_FUNCTION ROWCOUNT

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

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

Variable *nb-rows* INT(4) O number of rows affected by a query

Example 1

```
LOCAL ROWCOUNT%

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

SQL_EXEC NS_FUNCTION ROWCOUNT INTO :ROWCOUNT%
; if 10 rows correspond to this filter and then 10 rows have been
deleted
; then ROWCOUNT% will contain 10.
; if no row correspond to this filter then 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 occurrences = ', ROWCOUNT%
```

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

NS_FUNCTION SETBUFFERSIZE

Specifies the buffer size in number of rows for ARRAY FETCH operations and stored procedures with array parameters.

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

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

Notes

1. By default, *size-buffer* equals 1 from Oracle 8.x (and 20 for Oracle 7.x). In a situation where one of the columns selected is LONG type, the default size is always 1.
2. The maximum size is determined by the columns selected and the size of memory Oracle has authorized to handle arrays. The Oracle ARRAYSIZE and MAXDATA parameters let you set the size. (see the Oracle documentation for more information.)
3. This NS_FUNCTION optimizes network transfers by allowing an increase in the number of lines transferred during a FETCH.

Example 1

```
;segment S_COL1
;      int V_COL1
;endsegment

;segment S_COL2
;      cstring V_COL2(10)
;endsegment

;segment S_COL1_S
;      S_COL1 SEG[10]
;endsegment

;segment S_COL2_S
;      S_COL2 SEG[10]
;endsegment

LOCAL COL1%
LOCAL COL2$(10)
LOCAL HCOL1%, HCOL2%
Local int      I%(4)
```

```
sql_exec drop table JYM
SQL_EXEC CREATE TABLE JYM ( COL1 INTEGER , COL2 VARCHAR2(10) )
if sql_error% <> 0
    message 'error' , sql_errmsg$(sql_error%)
endif

; ---- Create segments
NEW S_COL1_S , HCOL1%
NEW S_COL2_S , HCOL2%

S_COL1_S(HCOL1%).SEG[0].V_COL1 = 1
S_COL2_S(HCOL2%).SEG[0].V_COL2 = 'Albert%%%'
S_COL1_S(HCOL1%).SEG[1].V_COL1 = 2
S_COL2_S(HCOL2%).SEG[1].V_COL2 = 'Bernard%%'
S_COL1_S(HCOL1%).SEG[2].V_COL1 = 3
S_COL2_S(HCOL2%).SEG[2].V_COL2 = 'DANIEL%%%'
S_COL1_S(HCOL1%).SEG[3].V_COL1 = 4
S_COL2_S(HCOL2%).SEG[3].V_COL2 = 'ROGER%%%'

    Move 4 to I%
    SQL_EXEC NS_FUNCTION SETBUFFERSIZE :I% ;
if sql_error% <> 0
    message 'error' , sql_errmsg$(sql_error%)
endif

    SQL_EXEC INSERT INTO JYM (COL1, COL2) \
        DESCRIBE (:COL1%, :COL2%) \
        VALUES (:HCOL1%, :HCOL2%)
if sql_error% <> 0
    message 'error' , sql_errmsg$(sql_error%)
endif

dispose HCOL1%
dispose HCOL2%
```


Example 2

```

;SEGMENT SNAME
;  CSTRING NAME(11)
;ENDSEGMENT

;SEGMENT BUF_NAME
;  SNAME EMP[10]
;ENDSEGMENT

sql_execstr 'CREATE OR REPLACE PACKAGE PACK_TEST AS\
    TYPE namec_arr is table of emp.ename%type index by binary_integer;\
    PROCEDURE IN_OUT(NAMEC_ARRAY OUT NAMEC_ARR, FETCH_SIZE IN OUT
INTEGER);\
    cursor c_in_out is select ename from emp;\
END PACK_TEST;'
if sql_error% <> 0
    message 'error PACKAGE' , sql_errmsg$(sql_error%)
endif
sql_execstr 'CREATE OR REPLACE PACKAGE BODY PACK_TEST AS \
PROCEDURE IN_OUT(NAMEC_ARRAY OUT NAMEC_ARR, FETCH_SIZE IN OUT INTEGER)
IS \
counter binary_integer;\
begin\
    if not c_in_out%isopen then\
        open c_in_out;\
    end if;\
    counter := 1;\
    loop\
        fetch c_in_out into namec_array(counter);\
        exit when c_in_out%notfound or counter = fetch_size;\
        counter := counter + 1;\
    end loop;\
    if c_in_out%notfound then \
        fetch_size := counter - 1;\
        close c_in_out;\
    end if;\
end;\
end pack_test;'

if sql_error% <> 0
    message 'error BODY' , sql_errmsg$(sql_error%)
endif
;then
LOCAL CSTRING Ename(11)
LOCAL Nbr_Rows%, i%
LOCAL h_nos%,h_names%,h_names2%,h_dats%,curs%
MOVE 10 TO Nbr_Rows%
NEW BUF_NAME, h_names%
MOVE SQL_OPENCURSOR% TO CURS%
SQL_EXEC NS_FUNCTION SETBUFFERSIZE :Nbr_Rows% USING CURS%
; With Oracle 8 you have to describe your parameters if you use a
; custom type in your procedure
; Bug 6765
SQL_EXEC SQL_PROC pack_test.in_out (:Ename OUT [VARCHAR2, 1],:Nbr_Rows%
IN [NUMBER, 0]) VALUES (:h_names%) USING CURS%
IF SQL_ERROR% <> 0
    Message 'error', SQL_ERROR%&`: '&SQL_ERRMSG$(SQL_ERROR%)
ENDIF

```

```
MOVE 0 TO i%
WHILE i% < Nbr_Rows%
  INSERT AT END 'Longueur' &&length BUF_NAME(h_names%).EMP[i%].NAME&& \
    'Valeur=' &BUF_NAME(h_names%).EMP[i%].NAME TO LISTBOX1
  MOVE i%+1 TO i%
ENDWHILE
SQL CLOSECURSOR
DISPOSE h_names%
```

Example 3

See the example of SQL_PROC function.

See also

SQL_PROC

The examples of "Array Processing", " Calling Stored Procedures and Stored Functions".

NS_FUNCTION STATEMENT

Retrieves the full statement used in the query sent to the SQL engine.

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

Variable *query-string* CSTRING O statement used in the query sent to the SQL engine

Notes

- 1.The INTO clause (even precised) is never traced.
- 2.The host variables are never traced and appeared on :A1 form.

Example

```
LOCAL VALUES$, SENTENCE$  
  
MOVE "HELLO" TO VALUES$  
SQL_EXEC SELECT COL1 FROM TABLE WHERE COL2=:VALUES$  
  
SQL_EXEC NS_FUNCTION STATEMENT INTO : SENTENCE$  
MESSAGE "the query is :", SENTENCE$  
  
SENTENCE$=SELECT COL1 FROM TABLE WHERE COL2='HELLO'
```

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 table is CHAR or VARCHAR2.

The TRIMCHARON mode allows to limit the size of the buffer in the network.

In TRIMCHAROFF mode, the blank spaces are kept.

Syntax **NS_FUNCTION TRIMCHAROFF**
 and
 NS_FUNCTION TRIMCHARON

Note

1. TRIMCHAROFF is the default mode.

Example

```
LOCAL C$, B$
; longstr is a varchar2(2000)
; & TEST CHAR(10)
SQL_EXEC INSERT INTO TOTO (ID, TEST, LONGSTR) VALUES (1, 'A234567890', \
\ 'lgstr23456789')
if sql_error% <> 0
    message 'error INSERT' , sql_errmsg$(sql_error%)
endif
SQL_EXEC INSERT INTO TOTO (ID, TEST, LONGSTR) VALUES (2, 'A2345', \
'lgstr2345    ')
if sql_error% <> 0
    message 'error INSERT' , sql_errmsg$(sql_error%)
endif
SQL_EXEC INSERT INTO TOTO (ID, TEST, LONGSTR) VALUES (3, 'A',
'lgstr      ')
if sql_error% <> 0
    message 'error INSERT' , sql_errmsg$(sql_error%)
endif
; Default mode
; ----- This loop will show <A234567890>
; <A2345                >
; <A                    >
; <lgstr23456789>,
; <lgstr234            >
; <lgstr                >
SQL_EXEC SELECT TEST, LONGSTR FROM TOTO
if sql_error% <> 0
    message 'error SELECT' , sql_errmsg$(sql_error%)
endif
INSERT AT END '{J[C],B[LIGHTRED]}DEFAULT' TO LISTBOX1
```

```

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

```

See also SQL_ERROR%, SQL_ERRMSG\$

NS_FUNCTION WARNINGSOFF, WARNINGSON

NS_FUNCTION WARNINGSOFF ignores warnings raised by the database. Practically, when SQL_ERROR% > 0, NS_FUNCTION WARNINGSOFF sets it to 0 to avoid warnings risking to stop the application.

NS_FUNCTION WARNINGSON allows to retrieve the warnings to be dealt with by the application.



This NS-FUNCTION is available from Oracle 8.0 onwards.

Syntax

NS_FUNCTION WARNINGSOFF

and

NS_FUNCTION WARNINGSON

Note

1. WARNINGSON is the default mode.

Example

```
local n#
SQL_EXEC select max(to_number(REEL1)) from AELTA into :n#
if sql_error% <> 0
    message 'Default' , sql_errmsg$(sql_error%)
endif
SQL_EXEC NS_FUNCTION WarningsOFF
IF SQL_ERROR% <> 0
    MESSAGE "Warning", SQL_ERROR% &&SQL_ERRMSG$(SQL_ERROR%)
endif
SQL_EXEC select max(to_number(REEL1)) from AELTA into :n#
if sql_error% <> 0
    message 'WarningsOFF' , sql_errmsg$(sql_error%); this warning is not
    raised
endif
SQL_EXEC NS_FUNCTION WarningsON
IF SQL_ERROR% <> 0
    MESSAGE "Warning", SQL_ERROR% &&SQL_ERRMSG$(SQL_ERROR%)
endif
SQL_EXEC select max(to_number(REEL1)) from AELTA into :n#
if sql_error% <> 0
    message 'WarningsON' , sql_errmsg$(sql_error%)
endif
```


Array Processing

Array FETCH

Notes

1. For each SQL query that contains a SELECT statement, the NSnnORxx.DLL library uses an internal buffer to retrieve the rows returned by Oracle. This ARRAY FETCH mechanism is transparent for the user.
2. By default, the ARRAY FETCH mechanism retrieves results in packets of 20 rows for Oracle 7.x and 1 row for Oracle 8.x. However, you can set the number of rows retrieved by the statement:

```
SQL_EXEC NS_FUNCTION SETBUFFERSIZE :nbLines%
```

Example

```
LOCAL A$,B$

SQL_EXEC SELECT COL1, COL2 INTO :A$ , :B$ FROM TAB1
; Here in one exchange between the client and the server
; we bring back 20 lines (default value for Oracle 7.x) on the client
in a
; buffer transparently for the user.

WHILE SQL_ERROR% = 0
    INSERT AT END 'A$=' & A$ & 'B$=' & B$ TO LISTBOX1
    SQL_EXEC FETCH
    ; Here as far as the 20 lines haven't been used, we access only in
the local
    ; buffer containing the 20 lines. Then when the buffer is empty, we
read the
    ; 20 lines again.
    ...
ENDWHILE
```

Array INSERT

Enables the insertion of data by packets of rows in one call.

Syntax

```
SQL_EXEC INSERT INTO    tablename (... , col_J, ...) \
                        DESCRIBE    ( ... , :host_J, ...) \
                        VALUES    ( ... , :handle_J, ...)
```

Notes

1. The data items to be inserted are stored in NCL segments whose handles are passed in the VALUES clause.
2. The DESCRIBE clause specifies the columns affected by the ARRAY INSERT operation using NCL variables with the same type and size as the items in the NCL segments that contain the data to be inserted into these columns.

Example

```
; ---- Declaration of segments
SEGMENT CODE
  INT COD(4)
ENDSEGMENT
SEGMENT NAME
  CSTRING NAM(20)
ENDSEGMENT
SEGMENT QUANT
  INT QUAN(4)
ENDSEGMENT

; ---- Declaration of segments arrays
SEGMENT CODES
  CODE SEG[10]
ENDSEGMENT
SEGMENT NAMES
  NAME SEG[10]
ENDSEGMENT
SEGMENT QUANTS
  QUANT SEG[10]
ENDSEGMENT

LOCAL HCODES%, H NAMES%, HQUANTS%
LOCAL INT      VCODE(4) ; These variables are the same type and
LOCAL CSTRING VNOM(20) ; same size as items of
LOCAL INT      VQTE(4)  ; NCL segments declared

; ---- CREATION OF THE TABLE
SQL_EXEC CREATE TABLE PRODUIT(CODE NUMBER(5), \
                                NOM VARCHAR2(20), \
                                QUANTITE NUMBER(8))

; ---- CREATION OF SEGMENTS
```

```
NEW CODES   , HCODES%
NEW NAMES   , HNAME%
NEW QUANTS  , HQUANT%

; ---- FILLING OF THE 7 FIRST LINES OF ARRAYS
MOVE 1       TO CODES(HCODES%).SEG[0].COD
MOVE 'STYLO'  TO NAMES(HNAME%).SEG[0].NAM
MOVE 152     TO QUANTS(HQUANT%).SEG[0].QUAN
MOVE 2       TO CODES(HCODES%).SEG[1].COD
MOVE 'AGENDA' TO NAMES(HNAME%).SEG[1].NAM
MOVE 152     TO QUANTS(HQUANT%).SEG[1].QUAN
...
MOVE 7       TO CODES(HCODES%).SEG[6].COD
MOVE 'GOMME'  TO NAMES(HNAME%).SEG[6].NAM
MOVE 285     TO QUANTS(HQUANT%).SEG[6].QUAN

; ---- NUMBER OF LINES USED IN ARRAYS
J% = 7
SQL_EXEC NS_FUNCTION SETBUFFERSIZE :J%

; ---- INSERTING 7 LINES IN ONE CALL
SQL_EXEC INSERT INTO PRODUIT(CODE,NOM,QUANTITE)\
        DESCRIBE (:VCODE   , :VNOM   , :VQTE)\
        VALUES  (:HCODES% , :HNAME% , :HQUANTS)
```

Array UPDATE

Enables the update data using packets of rows in one call.

Syntax	SQL_EXEC UPDATE	<i>tablename</i> \
	SET	<i>..., col_J = :host_J, ... \</i>
	WHERE	<i>... col_K = :host_K AND ... \</i>
	VALUES	<i>(... , :handle_J, ... , :handle_K, ...)</i>

Notes

1. The data items to be updated are stored in NCL segments whose handles are passed in the VALUES clause.
2. The columns affected by the array update operation are described by NCL variables with the same type and size as the items in the NCL segments that contain the data to be updated.

Example

```
SQL_EXEC UPDATE PRODUIT
SET      QUANTITE =:VQTE\
WHERE    CODE =:VCODE   \
VALUES   (:HQUANTS ,   :HCODES%)
```

Array DELETE

Enables the deletion of data by packets of rows in one call.

Syntax SQL_EXEC **DELETE FROM** *tablename* \
 WHERE ... *col_K* = :*host_K* AND ... \
 VALUES (... , :*handle_K* , ...)

Notes

1. The values permitting the selection of data items to be deleted are stored in NCL segments whose handles are passed in the VALUES clause.
2. The columns affected by the ARRAY DELETE operation are described by NCL variables with the same type and size as the items in the NCL segments that contain the data to be deleted.

Example

```
; ---- Delete all the products having the code contained  
;      in the handle's segment of HCODES%  
  
SQL_EXEC DELETE FROM PRODUIT \  
          WHERE    CODE =:VCODE \  
          VALUES   (:HCODES%)
```

Calling Stored Procedures and Stored Functions

Reminder

Oracle lets you call functions or stored procedures in two categories: simple (scalar) and/or array.



For more information, see the description of the SQL_PROC command.

➤ Syntax for Calling Stored Procedure

```
SQL_PROC [SCHEMA].PACKAGE_NAME.PROCEDURE_NAME
        {( ... , :host_I {mode [type,level]}}, ... , :host_J {mode [type,level]}}, ...)
        {VALUES ( ..., buffer_J, ...)}
```

SCHEMA corresponds by default to the user name who created the stored procedure/function.

➤ Syntax for Calling Stored Function

```
SQL_PROC :host_returned{mode[type,level]}=
[SCHEMA].PACKAGE_NAME.FUNCTION_NAME
        {( ... , :host_I {mode [type,level]}}, ... , :host_J {mode [type,level]}}, ...)
        {VALUES ( ..., buffer_J, ...)}
```

SCHEMA corresponds by default to the user name who created the stored procedure/function.

➤ SQL_PROC command parameters, description and values:

Parameters	Descriptions and values
<i>{}</i>	optional
<i>host_returned</i>	NCL variable for the return code of the function
<i>host_I</i>	SCALAR parameter: NCL variable
<i>host_J</i>	ARRAY parameter: NCL variable to find out the size and type of element in array
<i>buffer_J</i>	Buffer address containing a single column array associated with the <i>host_J</i> parameter.
<i>Mode</i>	IN, OUT, INOUT
<i>Type</i>	VARCHAR2, CHAR, CHARZ, STRING, MLSLABEL, VARCHAR, NUMBER, INTEGER, FLOAT, ROWID, DATE, RAW
<i>Level</i>	0: for a SCALAR parameter 1: for a ARRAY parameter

Steps Required

To call an Oracle stored procedure with array parameters, you need to carry out the following steps:

1. Define the corresponding segments for each array parameter.
2. Declare host variables with the same type and size as the items in the segments defined above.
3. Allocate segments (specify handles).
4. Specify the number of rows to be exchanged with the server. This is usually equal to the size of your buffers.
5. Call the stored procedure or function.



If the stored procedure accepts one or more PL/SQL arrays as parameters, each PL/SQL array corresponds to a NCL segment and the field is the same size and type as the parameter.

Restrictions

When PL/SQL is used with the Oracle DBMS, stored procedures are subject to a number of restrictions that you should bear in mind when you create PL/SQL packages and procedures :

- Oracle PL/SQL procedures cannot accept parameters with the LONG type.
- Stored functions cannot return LONG values.
- If a stored procedure has an array parameter whose type is TABLE OF DATE' an internal PL/SQL error will occur if it attempts to insert values into or select values from a DATE column.
- To insert values into or retrieve values from a DATE column, you need to specify the array's type as:
 - TABLE OF CHAR,
 - TABLE OF VARCHAR,
 - TABLE OF VARCHAR2.
- As result, the PL/SQL procedure must use:
 - The TO_CHAR function to select a DATE column into a TABLE OF CHAR parameter.
 - The TO_DATE function to insert a date into DATE column from a TABLE OF CHAR parameter.

**16
bits**

- The size of alpha-type host variables (CHAR, VARCHAR, etc.) cannot be greater than 255 characters.
- The size of host variable must be the same as the parameter passes in the procedure. The declaration of a string-type host variables must therefore be followed by their size: `LOCAL A$(15)`.



For more information about SQL and PL /SQL, see the Oracle manuals.

Examples

Example 1 : simple parameters

Suppose we have the following stored function in PL/SQL:

```
CREATE OR REPLACE PACKAGE PROC_SIMPLES IS
PROCEDURE ADD_EMP(DEPART_NO IN EMP.DEPTNO%TYPE,
                  EMP_NO      IN EMP.EMPNO%TYPE,
                  EMP_NAME    IN EMP.ENAME%TYPE,
                  EMP_JOB     IN EMP.JOB%TYPE,
                  EMP_MGR     IN EMP.MGR%TYPE,
                  EMP_SAL     IN EMP.SALARY%TYPE) ;
END PROC_SIMPLES;
\
CREATE OR REPLACE PACKAGE BODY PROC_SIMPLES IS
PROCEDURE ADD_EMP(DEPART_NO IN EMP.DEPTNO%TYPE,
                  EMP_NO      IN EMP.EMPNO%TYPE,
                  EMP_NAME    IN EMP.ENAME%TYPE,
                  EMP_JOB     IN EMP.JOB%TYPE,
                  EMP_MGR     IN EMP.MGR%TYPE,
                  EMP_SAL     IN EMP.SALARY%TYPE) IS
BEGIN
INSERT INTO EMP(EMPNO, ENAME, JOB, MGR, DEPTNO,SALARY)
VALUES (EMP_NO, EMP_NOM, EMP_JOB, EMP_MGR,
        DEPARTURE_NO, EMP_SAL) ;
END ADD_EMP ;
END PROC_SIMPLES;
\
```

The following NCL call to the ADD_EMP stored procedure adds a new employee to the EMP table.

```
LOCAL NO_EMP%, NO_DEPARTURE%, NO_MGR%, SAL_EMP%
LOCAL CSTRING NAME_EMP(12), JOB_EMP(20)

MOVE 'Scott TIGER'      TO NAME_EMP
MOVE 'Computer scientist' TO JOB_EMP
MOVE 900                TO NO_EMP%
MOVE 55                 TO NO_DEPARTURE%
MOVE 400                TO NO_MGR%
```



```

MOVE 11000                TO SAL_EMP£

SQL_EXEC SQL_PROC \
      PROC_SIMPLES.ADD_EMP(:NO_DEPARTURE%, :NAME_EMP, \
                          :JOB_EMP, :NO_MGR%, :SAL_EMP£)

```

Example 2 : simple parameters

For a procedure that returns values (one row only), displaying these values involves displaying the NCL variables passed as parameters.

Suppose we have the PL/SQL stored procedure, *Sel_Emp*, which accepts the employee number as input and returns the row that matches the employee's: name, number, job, department number, manager number and salary.

```

CREATE OR REPLACE PACKAGE PROC_SIMPLES IS
PROCEDURE SEL_EMP(DEPARTURE_NO OUT EMP.DEPTNO%TYPE,
                  EMP_NO      IN OUT EMP.EMPNO%TYPE,
                  EMP_NAME    OUT  EMP.ENAME%TYPE,
                  EMP_JOB     OUT  EMP.JOB%TYPE,
                  EMP_MGR     OUT  EMP.MGR%TYPE,
                  EMP_SAL     OUT  EMP.SALARY%TYPE);
END PROC_SIMPLES;
\
CREATE OR REPLACE PACKAGE BODY PROC_SIMPLES IS
PROCEDURE SEL_EMP(DEPART_NO OUT EMP.DEPTNO%TYPE,
                  EMP_NO      IN OUT EMP.EMPNO%TYPE,
                  EMP_NAME    OUT  EMP.ENAME%TYPE,
                  EMP_JOB     OUT  EMP.JOB%TYPE,
                  EMP_MGR     OUT  EMP.MGR%TYPE,
                  EMP_SAL     OUT  EMP.SALARY%TYPE) IS
BEGIN
  SELECT EMPNO, ENAME, JOB, MGR, DEPTNO, SALARY
  FROM EMP INTO EMP_NO, EMP_NAME, EMP_JOB,
               EMP_MGR, DEPARTURE_NO, EMP_SAL
  WHERE EMPNO=EMP_NO;
END SEL_EMP;
END PROC_SIMPLES;
\

```

The NCL call to the stored procedure SEL_EMP is coded as follows:

```

LOCAL NO_EMP%, NO_DEPARTURE%, NO_MGR%, SAL_EMP£
LOCAL CSTRING NAME_EMP(12), CSTRING JOB_EMP(20)

MOVE 900 TO NO_EMP%

SQL_EXEC SQL_PROC
      PROC_SIMPLES.SEL_EMP(:NO_DEPART%, :NAME_EMP, \
                          :JOB_EMP, :NO_MGR%, :SAL_EMP£)

IF SQL_ERROR% = 0
  MESSAGE 'Result', NO_EMP% && NAME_EMP && \
               JOB_EMP && NO_DEPT && NO_MGR && SAL_EMP£
ENDIF

```

Example 3 : Array type parameters

Suppose we have the following stored procedure:

```
PROCEDURE IN_OUT (DEPART      IN  EMP.DEPTNO%TYPE,
                  ARRAY_NAME OUT TAB_NAME,
                  ARRAY_NUM  OUT TAB_NUMBER,
                  SIZE_FETCH IN  INTEGER);
```

Here is how you call the IN_OUT stored procedure:

```
; ---- Two NCL segments have to be declared
;      they correspond to ARRAY_NAME and ARRAY_NUM types
;      of the stored procedure
SEGMENT NAME
  CSTRING ENAME(10) ; ARRAY_NAME of the procedure
ENDSEGMENT;

SEGMENT NO
  INT ENUM(4) ; ARRAY_NUM of the procedure
ENDSEGMENT ;

;---- Declaration of the NCL buffers
SEGMENT NAMES
  NAME EMP[10]
ENDSEGMENT

SEGMENT NOS
  NO EMP[10]
ENDSEGMENT

LOCAL H_NUM%, H_NAMES%, NB_LINES%, I%
; ---- Declaration of the host variables (these variables which are
;      the same
;      type and the same size of each item
;      of segments)
LOCAL INT      NO_DEPT%
LOCAL INT      ENO%
LOCAL CSTRING  ENAME(10)
LOCAL INT      NB_LINES%

; ---- Specification of the number of lines to exchange with
;      the server
MOVE 10 TO NB_LINES%
SQL_EXEC NS_FUNCTION SETBUFFERSIZE :NB_LINES%

; ---- Creation of buffers and recovering of their addresses
;      in handles
NEW NOS , H_NUM%
NEW NOMS, H_NAMES%

; ---- Call of the stored procedure
SQL_EXEC SQL_PROC
      IN_OUT ( :NO_DEPT%, :ENO%, :ENAME, :NB_LINES%) \
      VALUES ( :H_NUM%, :H_NAMES%)

; ---- Recovering results
MOVE 0 TO I%
IF SQL_ERROR% <> 0
```

```

MESSAGE ' APPEL_PROC', SQL_ERRMSG$(SQL_ERROR%)
ELSE
  WHILE ( (I% < NB_LINES%))
    INSERT AT END NOS(H_NUMS%).EMP[I%].ENUM      \
      && NAMES(H_NAMES%).EMP[I%].ENAME    \
    TO LISTBOX
    MOVE I%+1 TO I%
  ENDWHILE
ENDIF

```

Example 4 : INPUT Array Parameters

Create a global package that contains all the procedures, array types and cursors used:

```

-- Declare the PL/SQL array type used for employee names
-- Note that it is far better to define the
-- the array data type as TABLE.COL%TYPE ==>
-- this ensures that the type declaration is independent
-- of the physical structure

CREATE OR REPLACE PACKAGE PACK_TEST_IN_IN AS
TYPE NAME_ARR IS TABLE OF EMP.ENAME%TYPE
  INDEX BY BINARY_INTEGER;

-- Declare the PL/SQL array type used for employee numbers
TYPE NUMBER_ARR IS TABLE OF EMP.EMPNO%TYPE
  INDEX BY BINARY_INTEGER;

-- Declare a TABLE OF VARCHAR2 for inserting values into
-- DATE columns
TYPE DATE_ARR IS TABLE OF VARCHAR2
  INDEX BY BINARY_INTEGER;

-- IN_IN procedure: accepts a department number as input
-- together with three arrays of employee names, numbers and
-- hiring dates to be inserted into the EMP table.
-- It also receives the number of employees
-- passed in the arrays
PROCEDURE IN_IN(DEPART IN EMP.DEPTNO%TYPE,
  NAME_ARRAY IN NAME_ARR,
  NUM_ARRAY IN NUMBER_ARR,
  DATE_ARRAY IN DATE_ARR,
  FETCH_SIZE IN INTEGER);

END PACK_TEST_IN_IN;

-- Create the package body
CREATE OR REPLACE PACKAGE BODY PACK_TEST_IN_IN AS

PROCEDURE IN_IN(DEPART IN EMP.DEPTNO%TYPE,
  NAME_ARRAY IN NAME_ARR,
  NUM_ARRAY IN NUMBER_ARR,
  DATE_ARRAY IN DATE_ARR,
  FETCH_SIZE IN INTEGER) IS

```

```

-- Declare a counter used to index the arrays:
COUNTER BINARY_INTEGER;

BEGIN
-- Loop for the number of rows passed in each array
  FOR COUNTER IN 1..FETCH_SIZE LOOP
-- Insert values from the array into the EMP table
-- All employees inserted belong to the same department
-- (passed as an IN parameter to the proc.)
    INSERT INTO EMP (EMPNO, ENAME, DEPTNO, EMP_DATE)
      VALUES (NUM_ARRAY(COUNTER), NAME_ARRAY(COUNTER),
              TO_DATE(DATE_ARRAY(COUNTER), DEPART) );
  END LOOP;
END;
END PACK_TEST_IN_IN;
/

```

➤ *INIT event for the main window*

```

; ----- Declare NCL segments for the PL/SQL array parameters Segment
;           for the IN_IN procedures
SEGMENT SNAME
  CSTRING NAME(10)
ENDSEGMENT;
SEGMENT SNO
  INT NO(2)
ENDSEGMENT;
SEGMENT SDAT
  CSTRING DAT(10)
ENDSEGMENT

; NCL buffer
SEGMENT BUF_NO
  SNO EMP[10]
ENDSEGMENT
SEGMENT BUF_NAME
  SNAME EMP[10]
ENDSEGMENT;
SEGMENT BUF_DAT
  SDAT EMP[10]
ENDSEGMENT;
; Global declaration for segment and cursor handles
GLOBAL H_NOS%, H_NAMES%, H_DATS%, CURS%

; Load the DLL
SQL_INIT "NS02OR7"
IF SQL_ERROR% <> 0
  Message "INIT", SQL_ERRMSG(SQL_ERROR%)
ENDIF

; Open the database
SQL_OPEN "ORACLE", "SCOTT/TIGER@B:ORA7"
IF SQL_ERROR% <> 0
  Message "OPEN_DB", SQL_ERRMSG(SQL_ERROR%)
ENDIF

```

➤ **TERMINATE event for the main window**

```
SQL_CLOSE "ORACLE"
IF SQL_ERROR% <> 0
    MESSAGE "CLOSE" && SQL_ERROR%, SQL_ERRMSG$(SQL_ERROR%)
ENDIF
SQL_STOP
```

➤ **EXECUTED event for the IN_IN PushButton**

```
LOCAL CSTRING  ENAME(10)
LOCAL INT      ENUMBER(2)
LOCAL CSTRING  EDATE(10)
LOCAL DEPART_NO%
LOCAL NBR_ROWS%, I%

MOVE 40 TO DEPART_NO%
MOVE 8  TO NBR_ROWS%

;***** Create segment *****
NEW BUF_NAME, H_NAMES%
NEW BUF_NO, H_NOS%
NEW BUF_DAT, H_DATS%

;*** Fill NCL segment for insertion ****
MOVE "JONES"      TO BUF_NAME(H_NAMES%).EMP[0].NAME
MOVE 163          TO BUF_NO(H_NOS%).EMP[0].NO
MOVE "07-09-92"   TO BUF_DAT(H_DATS%).EMP[0].DAT

MOVE "DOE"        TO BUF_NAME(H_NAMES%).EMP[1].NAME
MOVE 158          TO BUF_NO(H_NOS%).EMP[1].NO
MOVE "05-11-89"   TO BUF_DAT(H_DATS%).EMP[1].DAT

MOVE 2 TO I%
WHILE I% < 10
    MOVE "ROBERTS"      TO BUF_NAME(H_NAMES%).EMP[I%].NAME
    MOVE I%             TO BUF_NO(H_NOS%).EMP[I%].NO
    MOVE "11-11-95"     TO BUF_DAT(H_DATS%).EMP[I%].DAT
    MOVE I%+1 TO I%
ENDWHILE

; ---- Specify number of rows
NB_ROWS% = 8
SQL_EXEC NS_FUNCTION SETBUFFERSIZE:NB_ROWS%

MOVE SQL_OPENCURSOR% TO CURS%

; ***** Call the stored procedure IN_IN *****
SQL_EXEC SQL_PROC
    PACK_TEST_IN_IN.IN_IN(:DEPART_NO%, :ENAME,
                          :ENUMBER, :EDATE, :NBR_ROWS%) \
    VALUES (:H_NAMES%, :H_NOS%, :H_DATS%) \
    USING CURS%
IF SQL_ERROR% <> 0
    MESSAGE "IN_IN", SQL_ERRMSG$(SQL_ERROR%)
ELSE
    MESSAGE "IN_IN", "OK"
```

```

ENDIF
SQL_CLOSECURSOR

DISPOSE H_NAMES%
DISPOSE H_NOS%
DISPOSE H_DATS%

```

Example 5 : OUTPUT Array Parameters

Create a global package that contains all the procedures, array types and cursors used:

```

-- Declare the PL/SQL array type used for employee names
-- Note that it is far better to define the
-- the array data type as TABLE.COL%TYPE ==>
-- this ensures that the type declaration is independent
-- of the physical structure
CREATE OR REPLACE PACKAGE PACK_TEST_IN_OUT AS
TYPE NAME_ARR IS TABLE OF EMP.ENAME%TYPE
    INDEX BY BINARY_INTEGER;

-- Declare the PL/SQL array type used for employee numbers
TYPE NUMBER_ARR IS TABLE OF EMP.EMPNO%TYPE
    INDEX BY BINARY_INTEGER;

-- Declare a TABLE OF VARCHAR2 for retrieving DATE values
TYPE DATE_ARR IS TABLE OF VARCHAR2
    INDEX BY BINARY_INTEGER;

-- Declare the cursor used by the "IN_OUT" procedure.
-- This cursor performs a SELECT on the employee names,
-- numbers and hiring dates for a given department
CURSOR C_IN_OUT(DEPART IN EMP.DEPTNO%TYPE) IS
    SELECT ENAME, EMPNO, TO_CHAR(EMP_DATE)
    FROM EMP
    WHERE DEPTNO = DEPART;

-- IN_OUT procedure: accepts a department number as input
-- and returns the names, numbers and hiring dates of the
-- employees in this department in each array
PROCEDURE IN_OUT(DEPART IN EMP.DEPTNO%TYPE,
    NAME_ARRAY OUT NAME_ARR,
    NUM_ARRAY OUT NUMBER_ARR,
    DATE_ARRAY OUT DATE_ARR,
    FETCH_SIZE IN OUT INTEGER);
END PACK_TEST_IN_OUT;
/
-- Create the package body
CREATE OR REPLACE PACKAGE BODY PACK_TEST_IN_OUT AS

PROCEDURE IN_OUT(DEPART IN EMP.DEPTNO%TYPE,
    NAME_ARRAY OUT NAME_ARR,
    NUM_ARRAY OUT NUMBER_ARR,
    DATE_ARRAY OUT DATE_ARR,
    FETCH_SIZE IN OUT INTEGER) IS

```

```

-- Declare a counter used to index the array(s)
Counter BINARY_INTEGER;

BEGIN
  IF NOT C_IN_OUT%ISOPEN
  THEN
    -- If the cursor associated with this procedure is not
    -- open, then this is the first call to the procedure ==>
    -- the Nat System interface requests the 1st data stream=>
    -- open the cursor
    OPEN C_IN_OUT(DEPART);
  END IF;

  -- Set the counter to the start of the arrays
  COUNTER:= 1;
  -- Potentially infinite loop
  LOOP
    -- Place each SELECT result row in each array
    FETCH C_IN_OUT INTO NAME_ARRAY(COUNTER),
                      NUM_ARRAY(COUNTER),
                      DATE_ARRAY(COUNTER);

    -- Exit conditions: If the requested number of rows
    -- has been read (fetch_size) or if there is nothing
    -- left to read (cursor%NOTFOUND)
    EXIT WHEN C_IN_OUT%NOTFOUND OR COUNTER = FETCH_SIZE;
    -- Stay in loop and increment counter
    COUNTER:= COUNTER + 1;
  END LOOP;
  -- Exit loop
  IF C_IN_OUT%NOTFOUND
  THEN
    -- Exit from loop because all the SELECT rows have been
    -- retrieved ==> start by closing the cursor then set the
    -- fetch size variable to the number of rows actually
    -- inserted into the arrays.
    -- This enables the Nat System interface to know how many
    -- items were returned.
    FETCH_SIZE:= COUNTER - 1;
    CLOSE C_IN_OUT;
  END IF;
  -- If all rows required have been read, there is nothing
  -- left to do
END;
END PACK_TEST_IN_OUT;
/

```

► INIT event for the main window

```

; ----- Global declaration for segment and cursor handles
GLOBAL H_NOS%,H_NAMES%, H_DATS, CURS%

; Load the DLL
SQL_INIT "NS02OR7"

IF SQL_ERROR% <> 0
  MESSAGE "INIT", SQL_ERRMSG(SQL_ERROR%)

```

```

ENDIF

; Open the database
SQL_OPEN "ORACLE", "SCOTT/TIGER@B:ORA7"
IF SQL_ERROR% <> 0
    MESSAGE "OPEN_DB", SQL_ERRMSG$(SQL_ERROR%)
ENDIF

```

➤ **TERMINATE event for the main window**

```

SQL_CLOSE "ORACLE"
IF SQL_ERROR% <> 0
    MESSAGE "CLOSE" && SQL_ERROR%, SQL_ERRMSG$(SQL_ERROR%)
ENDIF
SQL_STOP

```

➤ **EXECUTED event for the IN_OUT PushButton**

```

LOCAL CSTRING  ENAME(10)
LOCAL INT      ENUMBER(2)
LOCAL CSTRING  EDATE(10)
LOCAL DEPART_NO%
LOCAL NBR_ROWS%, I%

MOVE 40 TO DEPART_NO%
MOVE 8 TO NBR_ROWS%

;***** Create segment *****
NEW BUF_NAME, H_NAMES%
NEW BUF_NO, H_NOS%
NEW BUF_DAT, H_DATS%

;***** Open cursor *****
MOVE SQL_OPENCURSOR% TO CURS%

;*****Specify number of rows*****
NB_ROWS% = 8
SQL_EXEC NS_FUNCTION SETBUFFERSIZE:NBR_ROWS%

;***** Call the stored procedure IN_OUT *****
SQL_EXEC SQL_PROC
    PACK_TEST_IN_OUT.IN_OUT(:DEPART_NO%, :ENAME, \
                            :ENUMBER, :EDATE, \
                            :NBR_ROWS%) \
    VALUES (:H_NAMES%, :H_NOS%, :H_DATS%) \
    USING CURS%
IF SQL_ERROR% <> 0
    MESSAGE " IN_OUT ", SQL_ERRMSG$(SQL_ERROR%)
ELSE
    MESSAGE "IN_OUT", "OK"
ENDIF

;***** Retrieve results *****
MOVE 0 TO I%
WHILE I% < NBR_ROWS%
    MESSAGE "RESULT", BUF_NAME(H_NAMES%).EMP[I%].NO & \
    BUF_NO(H_NOS%).EMP[I%].NAME & \

```



```

                                BUF_DAT (H_DATS%) .EMP [I%] .DAT
    MOVE I%+1 TO I%
ENDWHILE
SQL_CLOSECURSOR
DISPOSE H_NAMES%
DISPOSE H_NOS%
DISPOSE H_DATS%

```

Example 6 : Calling a Stored Function

Suppose we have the following stored function in PL/SQL:

```

CREATE OR REPLACE PACKAGE SIMPLE_FUNCS IS
FUNCTION GET_EMP_NAME (V_EMP_NO IN INTEGER)
    RETURN EMP.ENAME%TYPE;
END SIMPLE_FUNCS ;
/
CREATE OR REPLACE PACKAGE BODY SIMPLE_FUNCS IS
FUNCTION GET_EMP_NAME (V_EMP_NO IN INTEGER)
    RETURN STATE.STA_NAME%TYPE IS
    NAME CHAR (20) ;
BEGIN
    SELECT ENAME INTO NAME FROM EMP
        WHERE EMPNO = V_EMP_NO;
    RETURN (NAME ) ;
END GET_EMP_NAME ;
END SIMPLE_FUNCS;
/

```

NCL call to the stored procedure ADD_EMP which adds a new employee to the table EMP Local Emp_No%.

```

LOCAL CSTRING EMP_NAME (20)

MOVE 900 TO EMP_NO%
SQL_EXEC \
    SQL_PROC:EMP_NAME = SIMPLE_FUNCS.GET_EMP_NAME (:EMP_NO)
IF SQL_ERROR% <> 0
    MESSAGE "ERROR", SQL_ERRMSG$ (SQL_ERROR%)
ELSE
    MESSAGE "RETURN VARIABLE", EMP_NAME
ENDIF

```



Calling stored functions with array parameters is similar to calling stored procedures with parameters. For more details, see the examples of stored procedures.

Example 7 : Calling in DescribeOn Mode

In **DescribeOn** mode, it's up to the programmer to specify the mode, type and level of the parameters used by the stored procedure.

Advantage: time savings, as the NSnnORxx library won't control the type of parameters in the procedure(s) stored in the database.

Inconvenience: the type of parameters in the stored procedure(s) must be identical to those of the database. Thus you lose the advantage of the "...%TYPE" functions (see Oracle manuals for more informations).

```
; ---- Change mode (change from default mode)
SQL_EXEC NS_FUNCTION DESCRIBEON

; ---- The user must specify the description of parameters
SQL_EXEC SQL_PROC \
    PACK_TEST_IN_OUT.IN_OUT(:DEPART_NO% IN    [INTEGER ,0],\
                           :ENAME          OUT  [VARCHAR2,1],\
                           :ENUMBER        OUT  [NUMBER  ,1],\
                           :EDATE          OUT  [DATE    ,1],\
                           :NB_ROW%        INOUT [INTEGER ,0]) \
    VALUES (:H_NAMES%, :H_NUMS%, :H_DATS%) \
    USING CURS%
```

XML Type

Oracle9i introduces a new type to handle a XML document for all the classical commands (SELECT), complete or partial extraction of documents, insertion and update of documents with validation (XMLSchema, DTD, or others).

With this new type, SQL/XML developers can validate the documents during insertion and recovery of blocks by Xpath directly in base without returning the whole document and passing through additional parsers like DOM, SAX or others. Then, you have better performances.

Syntax of XMLTYPE:

```
SYS.XMLTYPE {  
  
    CREATEXML (clob) !  
  
    CREATEXML (Varchar2) !  
  
    EXTRACT (xpath) !  
  
    EXISTSNODE (xpath) !  
  
    ISFRAGMENT  
  
    GETCLOBVAL !  
  
    GETSTRINGVAL ! (for the XML of less than 4000 characters)  
  
    GETNUMBERVAL  
  
    }
```

The XMLTYPE type is stored in base in a CLOB (Character Large Object), and has several functions to insert it and extract it, etc... in the same way that stored procedures which allows to validate and to handle XML data.

From now on, the Nat System products support the Oracle CLOB type as it does for Varchar2. In addition, Nat System inserts NCL new types to represent the hosts variables which will contain XML document:

- Type_sql_xml% : to handle XMLbuffers (for insertion primarily)
- Type_sql_select_xml% : to extract an XML content in a file.

- `Type_sql_insert_xml%` : to insert the content from a file.

*Examples:**Creation:*

```
;Create

SQL_EXEC CREATE TABLE TABXML (xml_col SYS.XMLTYPE, int_col INTEGER)
;insert Buffer
local xml$ (3000)
local nclvar hl[2]
local sql$

num% = 2
xml$ = '<name>DUPONT</name>'
hl[0].ptr_var = @xml$
hl[0].type_var = type_sql_xml% ;
hl[0].size_var = sizeof xml$
hl[1].ptr_var = @num%
hl[1].type_var = type_sql_int%
hl[1].size_var = 4

SQL$ = 'INSERT INTO TAB_XML (XML_COL, INT_COL) VALUES
(SYS.XMLTYPE.CREATEXML(:),
:)'

SQL_EXEC LONGSTR @SQL$, @HL, -1
message 'insert xml buffer' && sql_error%, sql_errmsg$ (sql_error%)
```

Inserting a file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
<servlet>
<servlet-name>WebCounter</servlet-name>
<servlet-class>FAQAnswerServlet</servlet-class>
</servlet>
<servlet>

<servlet-name>oracle-xsql-servlet</servlet-name>
<servlet-class>oracle.xml.xsql.XSQLServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>oracle-xsql-servlet</servlet-name>
<url-pattern> *.xsql </url-pattern>
</servlet-mapping>
</web-app>
```

Example:

```

local xmlFile$
local nclvar hl[2]
local sql$

num% = 1

XML$ = 'C:\..\MES_DOCUMENTS\WEBAPP.XML'
hl[0].ptr_var = @xmlFile$
hl[0].type_var = type_sql_insert_xml% ;
hl[0].size_var = sizeof xmlFile$
hl[1].ptr_var = @num%
hl[1].type_var = type_sql_int%
hl[1].size_var = 4

SQL$ = 'INSERT INTO TAB_XML (XML_COL, INT_COL) VALUES
(SYS.XMLTYPE.CREATEXML(:),
:)'

SQL_EXEC_LONGSTR @SQL$, @HL, -1
message 'insert xml file && sql_error%, sql_errmsg$ (sql_error%)

```

Return a complete xml file in a buffer of cstring, char, segment ... type:

```

LOCAL NUM%
LOCAL CHAR XML$ (2000)
LOCAL PATH$
LOCAL H%

NUM% = 1
XML$ = ''
;SELECT
SQL_EXEC SELECT T.XML_COL.GETCLOBVAL () FROM TABXML T WHERE \
T.INT_COL = :NUM%
WHILE (SQL_ERROR% = 0)
    SQL_EXEC FETCH INTO :XML$
    IF (SQL_ERROR% = 0)
        MESSAGE SQL_ERROR%, XML$
    ENDIF
ENDWHILE

```

Return a complete xml file in another file:

```

LOCAL NUM%
LOCAL XMLFILE$
LOCAL NCLVAR HL[2]
LOCAL SQL$

NUM% = 1
FILL XML$, 0, SIZEOF XML$
;XML$ = '<NAME>RACHEL</NAME>'
XMLFILE$ = 'C:\TEMP\EXEMPLE.XML'
HL[0].PTR_VAR = @XMLFILE$
HL[0].TYPE_VAR = TYPE_SQL_SELECT_XML% ;
HL[0].SIZE_VAR = SIZEOF XMLFILE$
HL[1].PTR_VAR = @NUM%
HL[1].TYPE_VAR = TYPE_SQL_INT%
HL[1].SIZE_VAR = 4

SQL$ = 'SELECT T.XML_COL.GETCLOBVAL () FROM TABXML T INTO : WHERE
T.INT COL = :'
SQL_EXEC_LONGSTR @SQL$, @HL, -1

```

```
MESSAGE 'SELECT XML FILE' && SQL_ERROR%, SQL_ERRMSG$ (SQL_ERROR%)
```

Return all the xml blocks referenced by an Xpath\$:

```
LOCAL NUM%  
LOCAL CHAR XML$ (2000)  
LOCAL PATH$  
LOCAL H%  
NUM% = 1  
XML$ = ''  
PATH$ = 'WEB-APP/SERVLET'  
SQL_EXEC SELECT T.XML_COL.EXTRACT(:PATH$).GETCLOBVAL () FROM TABXML T  
WHERE T.INT_COL  
= :NUM%  
SQL_EXEC FETCH INTO :XML$  
MESSAGE SQL_ERROR%, XML$
```

Result:

```
<servlet>  
  <servlet-name>WebCounter</servlet-name>  
  <servlet-class>FAQAnswerServlet</servlet-class>  
</servlet>  
<servlet>  
  <servlet-name>oracle-xsql-servlet</servlet-name>  
  <servlet-class>oracle.xml.xsql.XSQLServlet</servlet-class>  
</servlet>
```