ORACLE Driver Guide

UNIFACE

10109027205-00
Revision 0
Oct 1999
ORA

UNIFACE V7.2.05
ORACLE Driver Guide

Revision 0

24-hour online customer support

Pillars of Wisdom is an Internet-based support service which provides real-time access to a wealth of UNIFACE product and technical information. Features include online product documentation, technical tips and know-how, up-to-date platform availability and product fixes. You can obtain full access privileges for Pillars of Wisdom by completing an online registration form (customer license information is required) at the following address:
http://uniface.pillars.compuware.com/

Your suggestions and comments about UNIFACE documentation and course material are highly valued. Please send your reactions to:

Compuware Europe B.V.
Delivery Methods & Practices
P. O. Box 12933
1100 AX Amsterdam
The Netherlands

e-mail: DM&P-Hotline@nl.compuware.com
fax: +31 (0)20 311-6213

To order UNIFACE publications, contact your UNIFACE representative.

Contents

3 System environment

4 Configuring

5 sql and the SQL Workbench

14 Error messages

Chapter 1 Introduction

Table 1-1 ORACLE driver.

Version information	
DBMS version	7.3, 8
UNIFACE version	7.2
Three-letter mnemonic	ORA
UNIFACE driver version	U3.3, U3.5, U4.0, U5.0

This module describes versions U3.3, U3.5, U4.0, and U5.0 of the ORACLE driver. A complete description of compatibility with earlier driver versions and how to upgrade to the U3.*x*, U4.0, or U5.0 driver is provided in chapter 11 *Generic database conversion procedure*, and chapter 12 *Migrating between U2.x, U3.x, U4.0, and U5.0*.

The drivers and the ORACLE versions supported, are shown in table 1-2:

Table 1-2 Version supported.

UNIFACE driver	ORACLE version
U3.3	7.3
U3.5	8
U4.0	8
U5.0	8

The ORA U3.5 driver has the same functionality as the ORA U3.3 driver, but they are compiled against two different versions of ORACLE. The ORA U3.5 driver is compiled against ORACLE8.

The ORA U3.*x*, U4.0, and U5.0 drivers differ from each other as follows:

- In ORACLE7.3, using the U3.*x* driver, large objects were mapped to the Long or Long Raw data types. However, only one Long or Long Raw column could be held in any table. Using the U4.0 and U5.0 drivers, this restriction does not apply. The U4.0 and U5.0 drivers support large objects (LOBs) for segmented fields, using the ORACLE Character LOB (CLOB) and Binary LOB (BLOB) data types.
- ORA U3.5 is the 'compatibility driver'. ORA U3.5 allows you to migrate from ORACLE7 to ORACLE8 without needing to make any changes to the system.
- The U4.0 driver uses the ORACLE8 OCI functions. This affects any user accessing ORACLE from 3GL. See chapter 13 *Accessing ORACLE from 3GL* for more information.
- The U5.0 driver offers the same functionality as U4.0, but also enables UNIFACE to run in conjunction with a transaction manager using the XA Interface.

If you want to upgrade your applications and databases to driver version U3.*x*, U4.0, or U5.0, see chapter 12 *Migrating between U2.x, U3.x, U4.0, and U5.0* for more information.

1.1 Important notes on U3.*x*, U4.0, and U5.0 drivers

When using ORACLE7.*x* or 8, you should keep the following points in mind:

- There is no ORT variation of a U3.*x*, U4.0, or U5.0 driver. The functionality provided in the ORT variations of earlier drivers is included in the U3.*x* ORA driver.
- The U3.*x*, U4.0, and U5.0 drivers fully support the features in ORACLE7.
- The U3.3 and U3.5 drivers do not support the new features in ORACLE7.3 or ORACLE8. The U4.0 and U5.0 drivers incorporate the new ORACLE8 large object support.
- The U3.*x*,U4.0, and U5.0 drivers use the ORACLE7 architecture much more efficiently than the U2.1 driver.
- It is recommended that you use a U3.*x*,U4.0, or U5.0 driver for all new development, and that you upgrade current applications and databases to a U3.*x*, U4.0, or U5.0 driver. If you use or intend to use large objects and are using ORACLE8, it is recommended that you

use a U4.0 or U5.0 driver for new developments. Upgrading existing applications to use a U4.0 or U5.0 driver involves some work, as documented in chapter 12 *Migrating between U2.x, U3.x, U4.0, and U5.0*. If you are using ORACLE8 but not using large objects, the U4.0 and U5.0 drivers offer no advantages over the U3.5 driver.

- You can use a U3.*x* driver on a database which previously used the U2.1 driver, without performing an upgrade. You must, however, first set a number of U3.*x* driver options and modify the existing assignment files. If you do not do this, your application is unable to use some of the new features provided by a full upgrade. For information on this procedure, and how to fully upgrade to the U3.*x* driver, see chapter 12 *Migrating between U2.x, U3.x, U4.0, and U5.0*.
- The U5.0 driver should be used *only* if you plan to use a transaction manager. If you do not plan to use a transaction manager, use the U4.0 driver. Since the XA interface is available only on ORACLE servers, do not use the U5.0 driver on an ORACLE client. It could cause errors or crashes.

1.2 Features and enhancements

The U3.*x* drivers support many of the features introduced in ORACLE7. As mentioned previously, the U3.5 driver is compiled against ORACLE8, but adds no new functionality while the U4.0 and U5.0 drivers offer ORACLE8 LOB support.

1.2.1 U3.*x* drivers

UNIFACE segmented fields are mapped to ORACLE Long and Long Raw data types. This means that both UNIFACE and non-UNIFACE tools can access very large ORACLE fields containing text or binary data. Other large fields are implemented using overflow tables containing Long and Long Raw fields.

1.2.2 U4.0 and U5.0 drivers

UNIFACE segmented fields are mapped to ORACLE CLOB and BLOB data types. This means that both UNIFACE and non-UNIFACE tools can access very large ORACLE fields containing text or binary data. Similarly to the U3.*x* drivers, other large fields are implemented using overflow tables containing Long and Long Raw fields.

The U5.0 driver enables the use of transaction managers with UNIFACE.

1.2.3 Stored packages

The driver creates stored packages which it uses to improve performance when accessing the database.

1.2.4 Statement caching and cursor management

Statement caching and cursor management were completely redesigned for the U3.*x* driver to improve the performance involved in statement parsing and reduce client/server communication overhead.

Note: Overhead, as it is used in this document, refers to resources (usually processing time) consumed for purposes which are incidental, but necessary.

The U4.0 and U5.0 drivers retain the U3.*x* design as much as possible, but replace cursor management with handle and descriptor management. The U3.*x*, U4.0, and U5.0 drivers have been made more efficient in the following ways:

- The driver maintains a cache of the most recently used SQL and PL/SQL statements. This reduces both the number of parse calls and improves the client/server performance.
- The driver uses the ORACLE combined `execute/fetch` feature.
- Unnecessary rebinding of placeholders is avoided wherever possible.
- When stored packages are used, client/server communication is reduced considerably. A stored package requires the client/server overhead of only one statement, but it contains many I/O requests.
- The query generated for the `selectdb` Proc statement is cached.
- Deferred parsing is used.

You can see significant performance improvements when:

- SQL*Net is used.
- Your application accesses many different tables with a variety of I/O requests.

1.2.5 Improvements in referential integrity implementation

Both the lookup Proc statement and the existence check used for the implementation of referential integrity are more efficient in the current drivers than the U2.*x* drivers. This means that data transport from the ORACLE server to the ORA driver no longer occurs.

1.2.6 More storage formats supported

Both the variable-length Varchar2 and the fixed-length Char storage formats are now supported using the VC/VU and the C/U packing codes, respectively.

1.2.7 ORACLE wildcard characters in retrieve profiles

The driver uses the ESCAPE SQL clause with the LIKE operator to force the correct interpretation of ORACLE wildcard characters which have been used as literal characters in UNIFACE retrieve profiles.

1.2.8 Mapping of candidate keys

UNIFACE candidate keys are mapped to the Unique Key constraint instead of to unique indexes.

1.2.9 Number of concurrent logon paths

The maximum number of concurrent logon paths to ORACLE has been increased from four to a theoretical maximum of 32,767.

1.2.10 Use of ORACLE array fetching

ORACLE array fetching is now used wherever possible. Using it improves the client/server communication performance. This is particularly important when using SQL*Net. You can configure the array size using the driver options.

1.2.11 Step size configuration

You can now use a driver option to configure the step size of the UNIFACE stepped hitlist mechanism.

1.2.12 Support for ORACLE automatic logons

UNIFACE applications can now log on to ORACLE using the operating system user identification, instead of supplying a user name and password in the UNIFACE logon path information.

1.2.13 3GL services

Using the U3.*x* driver, your user-defined 3GL can access ORACLE using the Pro*C precompiler interface on the same logon path as the ORA driver. You can use both the Pro*C precompiler and the OCI interface in one application to access ORACLE.

Using the U4.0 and U5.0 drivers, you can use the OCI interface to access ORACLE on the same path as the ORA driver, but not the Pro*C precompiler.

1.2.14 Other features and enhancements

As well as those already mentioned in this section, the following features and enhancements are now available:

- The driver generates the optimizer hint `FIRST_ROWS` whenever appropriate.

- You can now change the value of the NLS_DECIMAL_CHARACTERS parameter within ORACLE.
- A value can be returned from PL/SQL statements in the sql Proc instruction and the SQL Workbench, but only if it is possible to convert it to the Varchar2 storage format.
- UNIFACE encloses identifiers in SQL and PL/SQL statements in double quotation marks ("). This allows you to use ORACLE reserved words for entity and field names in the application model, and it improves the portability of UNIFACE applications across DBMSs. For example, you may have developed your application with SYBASE, without paying attention to the ORACLE reserved words. It is recommended, however, that you avoid using ORACLE reserved words wherever possible.

1.3 ORACLE products supported

UNIFACE supports the ORACLE RDBMS, as well as the database options shown in table 1-3:

Table 1-3 ORACLE options supported by UNIFACE.

ORACLE product	Required	Supported	Recommended
Procedural option	No	Yes	Yes
Distributed option	No	Yes	No
Parallel server	No	Yes	No

UNIFACE has the following extra functionality when it is connected to an ORACLE server with the Procedural option:

- UNIFACE creates and uses stored packages which improve performance.
- PL/SQL (including stored procedure calls) can be used in the SQL Workbench and in the sql Proc instruction.

The functionality enabled by the Distributed option and the Parallel server option is transparent to UNIFACE.

Both version 1 and version 2 of SQL*Net are supported by UNIFACE.

1.3.1 The Pro*C product

You must install the Pro*C product before you can use UNIFACE with ORACLE. ORACLE7.3 and 8 use Pro*C 2.2.*x*. In particular, you should make sure that the following are present:

- The Pro*C libraries to which UNIFACE links.
- The link utility or link description files for the operating system (for example, **`loutl.com`** for Alpha OpenVMS and makefiles on UNIX). The UNIFACE installation program analyzes these files to determine the link information it needs.

User-defined 3GL can use both the precompiler interface and the ORACLE Call Interface (OCI).

*Note: As described above, you must have the Pro*C libraries and the relevant link description files before you can use UNIFACE with ORACLE7. It is not necessary, however, to install the Pro*C precompiler interface. See your ORACLE documentation for more information on how to achieve this.*

1.4 ORACLE products not supported

The following ORACLE products are not supported by UNIFACE:

- SQL*Module 3GL
- SQL*Connect
- Trusted ORACLE

Chapter 2 Installation

2.1 SQL scripts for creating the Application Objects Repository

The SQL scripts shown in table 2-1 are available in the UNIFACE installation directory and can be used to create the Repository in ORACLE:

Table 2-1 SQL scripts for creating the Repository in ORACLE. ***part 1 of 2***

Application Model	SQL script	Description
DICT	**ora3xdt.sql**	Create table and procedures
DICT	**ora3xdc.sql**	Create referential integrity
DICT	**ora3xdv.sql**	Verify referential integrity
DICT	**ora3xdd.sql**	Drop referential integrity
PRINTER	**ora3xpt.sql**	Create table and procedures
PRINTER	**ora3xpc.sql**	Create referential integrity
PRINTER	**ora3xpv.sql**	Verify referential integrity
PRINTER	**ora3xpd.sql**	Drop referential integrity
SYSENV	**ora3xst.sql**	Create table and procedures
SYSENV	**ora3xsc.sql**	Create referential integrity
SYSENV	**ora3xsv.sql**	Verify referential integrity
SYSENV	**ora3xsd.sql**	Drop referential integrity
TEXT	**ora3xtt.sql**	Create table and procedures
TEXT	**ora3xtc.sql**	Create referential integrity
TEXT	**ora3xtv.sql**	Verify referential integrity

Table 2-1 SQL scripts for creating the Repository in ORACLE. ***part 2 of 2***

Application Model	SQL script	Description
TEXT	**ora3xtd.sql**	Drop referential integrity
UVCS	**ora3xut.sql**	Create table and procedures
UVCS	**ora3xuc.sql**	Create referential integrity
UVCS	**ora3xuv.sql**	Verify referential integrity
UVCS	**ora3xud.sql**	Drop referential integrity

You *cannot* use these files when one of the following driver options is set because the Repository does not support these mappings:

- u2 default mapping
- u2 enhanced_mapping
- u2 enhanced_mapping_2

If you use **ora3xdt.sql** with the following driver options set, you must review and edit the file as described:

- disable packages–Remove the CREATE or REPLACE PACKAGE statements from the file, or just ignore the packages. If the ORACLE server does not have the Procedural Database option, you *must* remove the CREATE or REPLACE PACKAGE statements.
- map fixed length to variable–Change all Char columns to Varchar2.
- disable hint first_rows–Remove the hint from the package bodies in the file, or just ignore the hints.

You may set all other driver options when using **ora3xdt.sql**.

The **ora3xloa.sql** is also provided to create a view for the implementation of the Load Definitions utility. See section 2.2.3 *Load Definitions utility* for more information.

2.2 Utilities

2.2.1 Create Table utility

When you run the Create Table utility (Deployment–>Database Utilities–>Create Table), it generates an SQL script to create tables, indexes, stored package specifications, and stored package bodies. Scripts generated by the Create Table utility are correctly formatted for use with the ORACLE utilities SQL*Plus and SQL*DBA.

The effect of driver options

When the driver option `disable packages` is set, the SQL script generated by the Create Table utility does not create stored package specifications and stored package bodies. Use this option when generating SQL scripts for ORACLE servers without the Procedural option.

When the driver option `upgrade packages` is set, the generated SQL script creates only the stored package specifications and stored package bodies, and does not create any other schema objects. This option is set when performing the package upgrade procedure. See section 6.5 *Upgrading the stored packages*, for more information on the package upgrade procedure.

Note: Schema refers to the database design, that is, its structure. UNIFACE uses an entity schema.

2.2.2 Create Script utility

The Create Script form can be used to generate the SQL scripts required to perform the following:

- Create the DBMS referential integrity controls.
- Drop the DBMS referential integrity controls.
- Verify the referential integrity by checking for unlinked foreign keys which are not NULL.

For more information on the Create Script form, see the UNIFACE online help.

Note: The Create Script utility does not perform a referential integrity check on cascading and restricted relationships that check the relationship constraints. This is because ORACLE checks referential integrity constraints when the relationships are created.

2.2.3 Load Definitions utility

The Load Definitions utility (Goto Administration–>Exchange Models–>Load Definitions) loads the definitions of entities, fields, and keys.

By default with the ORA driver, the Load Definitions utility loads tables, but not views. If you want to load both tables and views, do the following:

1. Run the **ora3xloa.sql** script, which is available in the installation directory. The script creates a view which enables UNIFACE to load both tables and views. This view must be present in every ORACLE schema on which you want to run the Load Definitions utility. You can accomplish this in one of three ways:
   - Create the view in every schema.
   - Create the view in one schema, and create a public synonym for the view.
   - Create the view in one schema, and create private synonyms for the view in the schemas on which you want to run the Load Definitions utility.
2. Add the following line to the **idf.asn** assignment file:

   ```
   USER_TABLES.ORA6 = UNIFACE_TABLES.*
   ```

   Even though ORA6 is used in the line above, this line is applicable both to ORACLE6 (with the U2.0 ORA/ORT driver) and to ORACLE7.3 and 8. You may also specify this assignment in the **usys.asn** assignment file in the **USYS** directory (or **psys.asn** in the **PSYS** directory or **psv.asn** when you are accessing ORACLE via PolyServer).
3. Ensure that the logon path $ORA is defined in the assignment file. This is necessary when loading the schema objects of the user specified in this logon path.
4. Run the Load Definitions utility on the logon path $ORA.

The Load Definitions utility loads tables as modifiable entities, and it loads views as non-modifiable entities. However, some views are modifiable. For this reason, you should review the application model which is loaded by the Load Definitions utility. See section 3.3.8 *Views*, for more information on modifiable views.

Mapping ORACLE storage formats

The Load Definitions utility maps ORACLE storage formats to UNIFACE data types and packing codes as shown in table 2-2:

Table 2-2 How the Load Definitions utility maps ORACLE storage formats.

ORACLE	UNIFACE data type, packing code
Char	SS, Cn
Varchar2	SS, VCn
Varchar	SS, VCn
Number	N, Cn (Precision defined, but no scaling)
	N, Ci.j (Precision and scaling defined)
	F, F8 (No precision or scaling defined)
Date	E, E
Long	SS, SC*
Long Raw	R, SR*
Rowid	Not supported
Mlslabel	Not supported
Raw Mlslabel	Not supported

Fields which have the data type SS will also be formatted with a syntax of FUL. This means that multibyte characters can be included in these fields.

If you are not using the U3.*x* default packing code mapping, you should review the application model which is loaded by the Load Definitions utility. See section 8.2 *Modify packing code mapping*, for more information on the different ways packing codes can be mapped by the U3.*x* driver.

Loading primary keys, candidate keys, and indexes

The Load Definitions utility loads primary keys in different ways depending on the defined constraints. The primary keys are loaded as follows:

- If ORACLE primary key constraints are defined, they are always loaded as primary keys, regardless of whether the ORACLE primary key constraint is in an enabled or disabled state.
- If no primary key constraint is defined, but one or more unique indexes or enabled unique key constraints are defined, one of them is arbitrarily chosen to be the primary key.
- If there are no primary key or enabled unique key constraints, and no unique indexes defined in ORACLE, no primary key is loaded.

ORACLE unique indexes and enabled unique key constraints are loaded as candidate keys by the Load Definitions utility. The only exception to this is when a unique index or an enabled unique key constraint has already been chosen to be the primary key, as described above. Disabled unique key constraints are not loaded.

ORACLE nonunique indexes are loaded as indexes by the Load Definitions utility.

Chapter 3 System environment

3.1 Networking considerations

You can use either PolyServer or SQL*Net for network connectivity. Your choice of SQL*Net V1 or SQL*Net V2 is transparent to UNIFACE. See section 4.2.2 *Logon path specification* for more information on SQL*Net logon path specifications. See the *UNIFACE Configuration Guide* and the *UNIFACE Reference Manual* for more information on PolyServer logon path specifications.

SQL*Net

If you use SQL*Net, you must consider the following points to avoid problems arising from different data representations on machines with different architectures:

- Use the default packing code mapping of the U3.*x* driver, or set the driver option `u2 enhanced_mapping_2`. Do not specify the driver options `u2 default mapping` or `u2 enhanced_mapping`.
- Install UNIFACE with a system-independent byte sequence. For more information see the *Installation Guide* for your environment.
- It is recommended that you define the ORACLE client character set. It is possible that the default character set defined in your ORACLE client environment is different from the database to which your application connects. In this case, the character set in your ORACLE client environment should be set to `NLS_LANG`.

For example, you should use the NLS_LANG environment variable whenever the database to which your application connects uses a different character set than the default set defined in your ORACLE client environment. See section 4.8 *ORACLE character sets* for more information on how to specify a character set.

3.2 Necessary permissions

When developing a UNIFACE application in ORACLE, the developer must have the privileges to connect to the database, to create tables and to create procedures. These privileges are necessary both when schema objects are created 'on the fly', and when scripts generated by the Create Table utility are executed.

Once you have finished developing your application model, and all tables and packages have been created, you no longer need the CREATE TABLE and CREATE PROCEDURE privileges. If the ORACLE RDBMS does not have the Procedural option, or you set the disable packages driver option, the CREATE PROCEDURE privilege is not required. It is, however, recommended.

Your Database Administrator (DBA) should create a UNIFACE_DEVELOPER role, which is granted to all users working with the UNIFACE development environment. This role can be created by issuing the following SQL statements:

```
CREATE ROLE UNIFACE_DEVELOPER;
GRANT CREATE SESSION, CREATE TABLE, CREATE PROCEDURE TO UNIFACE_DEVELOPER;
GRANT UNIFACE_DEVELOPER TO user_name1, user_name2;
```

where *user_name1* and *user_name2* are the users who will be granted the same privileges as the UNIFACE_DEVELOPER role. It is recommended that you grant the CREATE VIEW, CREATE SEQUENCE, ALTER SESSION, CREATE TRIGGER, CREATE DATABASE LINK and CREATE SYNONYM privileges to the UNIFACE_DEVELOPER role.

3.3 Records, indexes, tables, and fields

UNIFACE can access and create the following ORACLE schema objects:

- Tables with columns and constraints
- Indexes
- Stored package specifications and stored package bodies

UNIFACE can also access views which have been defined in the application model.

Table 3-1 shows the maximums which apply to a UNIFACE application in ORACLE:

Table 3-1 UNIFACE application maximum values in ORACLE.

What	Maximum	Comments
Fields per record	254 fields	UNIFACE only supports 253 fields in modifiable tables.
Length of record	Unlimited	A UNIFACE record (not including segmented fields) has a maximum of 8192 bytes.
Fields per key	16 fields	
Number of indexes	Unlimited	
Maximum length of key	Unlimited	A UNIFACE record has a maximum of 8192 bytes.
Maximum length of a field	Not applicable	Dependent on the storage format.
SQL statement length	64 kilobytes	UNIFACE has a maximum of 10 kilobytes, except for the Create Table utility.

3.3.1 Location of schema objects

UNIFACE creates and accesses all schema objects[1] in the schema of the user specified in the logon path, unless you specify otherwise. UNIFACE allows you to access schema objects in another schema.

1. The 'schema' referred to is the ORACLE schema.

This can be accomplished by:

- Setting up synonyms in the schemas which are accessed by UNIFACE, or
- Setting up public synonyms.

Note: These are the only ways that you can access schema objects in another schema when using UNIFACE.

3.3.2 Naming rules for schema objects

The following rules apply when naming schema objects:

1. The maximum length for identifiers in ORACLE is 30 characters. When UNIFACE constructs object names, it automatically truncates entity and field names from the application model to comply with the ORACLE limit. For example, index number 11 of the table named `LONG_NAME_OF_THIRTY_CHARACTERS`, is named by UNIFACE as `LONG_NAME_OF_THIRTY_CHARACTI11`. UNIFACE uses this truncation method when it constructs names for the following objects:
   - Tables
   - Columns
   - Primary and Unique key constraints
   - Indexes
   - Package specifications and package bodies
   - Procedure parameters
   - Placeholders

   Ensure that the first 26 characters of all the entity and field names are unique, because they will be truncated somewhere after that point to comply with the ORACLE restriction.
2. Letters, numbers, and underscores are all allowed in entity and field names in the application model, however, the first character must be a letter. If you want to use an object which has a mixed-case name, you must create a synonym for that object in ORACLE.

For example, if you want to access an ORACLE table called `Dept`, use the name `DEPT` in the application model and create a synonym by issuing the following statement in the SQL Workbench:

```
CREATE SYNONYM DEPT FOR "Dept"
```

3. UNIFACE encloses identifiers in SQL and PL/SQL statements in double quotation marks (""). This allows you to use ORACLE reserved words for entity and field names in the application model, and it improves the portability of UNIFACE applications across DBMSs.

It is recommended that you avoid using ORACLE reserved words wherever possible.

3.3.3 Primary and candidate keys

Table 3-2 Rules for keys.

Rules for keys	Required by DBMS
Must be contiguous	No
Primary and candidate keys allowed to overlap	Yes
Primary key mandatory	No (but *is* required by UNIFACE)
Candidate keys mandatory	No

Primary Key constraints

The U3.*x* driver creates a named Primary Key constraint for the primary key defined in the application model. It does this when creating a table with the Create Table utility *and* when creating one on the fly. The constraint name is:

table_name P*index_number*

where *table_name* is the name of the table concerned, and *index_number* is the identification number of the index in UNIFACE. For a primary key in UNIFACE, *index_number* is always one.

The driver creates a Unique Key constraint for every candidate key in the application model. The constraint name is:

table_name C*index_number*

where *table_name* is the name of the table concerned, and *index_number* is the identification number of the index in UNIFACE. UNIFACE candidate key fields are mandatory, but the ORACLE Unique Key constraint does not enforce the Not Null constraint on the fields in the Unique Key. For this reason, UNIFACE creates the Not Null constraint for all candidate key fields which are not also in the primary key.

Storage formats allowed for key fields

The Char, Varchar2, Varchar, Date, Number, and Raw storage formats are allowed for key fields.

3.3.4 Indexes

UNIFACE creates an ORACLE index for every index in the application model. The ORACLE indexes are named as follows:

*table_name*I*index_number*

where *table_name* is the name of the table concerned, and *index_number* is the identification number of the index in UNIFACE.

Storage formats allowed for index fields

The Char, Varchar2, Varchar, Date, Number, and Raw storage formats are allowed for index fields.

3.3.5 Tables

Table creation

UNIFACE creates tables, columns, and both Primary and Unique Key constraints in ORACLE. This happens both on the fly and when you use the Create Table utility. It is recommended that you use the Create Table utility, and then the ORACLE SQL*Plus or SQL*DBA utility to execute the generated script.

Before accessing a table, UNIFACE first checks whether it exists by parsing one of the following queries:

SELECT *field_name1*, *field_name2*, ... FROM *table_name*

or,

SELECT 0 FROM *table_name*

The first query is used if you have not set the disable checks option in the USYS$ORA_PARAMS assignment. UNIFACE checks whether the table exists, *and* also checks whether the ORACLE storage formats of all columns are consistent with the information in the application model and driver options which change mapping behavior. This is less efficient than the second query, which is used if you have set the disable checks option in the USYS$ORA_PARAMS assignment. These queries are only parsed; they are not actually executed, and no data is fetched.

If UNIFACE finds that an expected table does not exist, it creates the table with all columns, Primary Key and Unique Key constraints, indexes and, optionally, the associated package specification and package body. If UNIFACE finds that the table exists, but some of the column storage formats are incorrect, it generates an ORA driver error.

For example, assume you have an ORACLE table called MYTABLE with a column called S_C, which has data type S and packing code C. This table was originally created with default driver options and now you attempt to access it with the driver option map fixed length to variable set. The driver generates the following error:

```
ORACLE Driver Error [-80]: Column has incorrect ORACLE storage format:
Table MYTABLE, Column S_C, expected storage format VARCHAR2, actual storage
format is CHAR.
```

Overflow tables

If UNIFACE variable-length techniques are used in an entity, both a base table and an overflow table are created. The name of the base table is the name of the entity in the application model (or the name assigned by means of entity assignment in the assignment file). The name of the overflow table is the name of the base table, prefixed by the character O. This means that when you are using UNIFACE variable-length techniques, you cannot define two entities where the name of one is the name of the other prefixed by the letter O.

Note: The name of an overflow table is the name of the base table prefixed by an O. If, however, the entity name contains one or more dollar signs ($) or hash marks (#) (which is only possible via entity assignment, but not in the application model), the O is inserted directly after the last $ or #. For example, with the entity assignment `MYENTITY.MYSCHEMA = $ORA:NEW$NAME`*, the base table is named* `NEW$NAME` *and the overflow table is named* `NEW$ONAME`*.*

The primary key in the overflow table is made up of the primary key fields from the base table and a field identifying the overflow segment.

3.3.6 Fields

Mandatory fields are created with the Not Null constraint, unless they are part of the primary key. This is because Primary Key fields in ORACLE implicitly have the Not Null constraint.

The way UNIFACE packing codes are mapped to ORACLE storage formats is described in chapter 8 *Data types and packing codes*.

3.3.7 Definition dependency

The definitions of ORACLE tables and indexes are taken from the definitions of entities in the UNIFACE application model. If you change the application model entity definitions after the ORACLE tables and indexes have been created, or after the generation of the SQL script to create those tables and indexes, you may have to re-create or alter those tables and indexes. For this reason, you must keep track of the dependencies between the UNIFACE application model and the schema objects in ORACLE.

3.3.8 Views

UNIFACE can access views in ORACLE, if those views have been correctly described in the application model. You must use an appropriate combination of UNIFACE data type and packing code to describe the storage format of every column in the view. This is particularly important when a select list expression in a view is an SQL function.

For example, substrings returned from the SUBSTR( ) function are fixed-length strings returned with trailing spaces. Such a select list item is equivalent to a Char column, and *not* to a Varchar2 column. Alternatively, you can use the RTRIM( ) SQL function in the view to remove trailing spaces and describe the select list item as a Varchar2 column.

When you define a view in the application model, you must make sure that the fields you define have the same names as in the alias list of the ORACLE view definition. See your ORACLE documentation (the CREATE VIEW statement) for more information. If a view is not modifiable, you must mark the entity as non-modifiable in the application model.

UNIFACE uses Rowids with modifiable entities. If a view is defined as a join expression or it is defined using a set operator (for example, UNION), the Rowid pseudo column cannot be selected. In that case, you must mark the view as non-modifiable in the application model. This is not an extra limitation, because such views are not modifiable in ORACLE. If you do not mark a view with a join expression or set operator as non-modifiable in the application model, UNIFACE attempts to use Rowids, and ORACLE generates the following error:

```
ORA-1445: cannot select ROWID from view with more than one table
```

UNIFACE uses stored packages with modifiable entities (provided the entity meets a number of other requirements, as described in chapter 6 *Stored packages*).

If a view is modifiable, you can choose between the following alternatives:

- Create the associated package for the view. For this purpose, you must first describe the entity in the application model, set the driver option upgrade packages, and run the Create Table utility for the entity describing the view.
- Set the driver option ignore missing packages.
- Set the driver option disable packages.

Chapter 4 Configuring

4.1 Driver assignments for UNIFACE system information

ORACLE is a good database management system to use for UNIFACE system information. In other words, $IDF, $UUU, and $SYS can all be assigned to ORACLE.

You must, however, keep the following considerations in mind when storing the Application Objects Repository (Repository) in ORACLE:

- Use the SQL scripts provided with the installation kit to create the Repository in ORACLE. For information on building the Repository, see section 2.1 *SQL scripts for creating the Application Objects Repository*.
- The dictionary uses variable-length storage techniques with platform-specific data formats. For this reason, you must install UNIFACE with the `independent byte order` option set if you want to access the dictionary via SQL*Net.
- You may store the Repository in ORACLE only when the default packing code mapping of the U3.*x* driver is used. The driver option `disable segmented fields` may be set, as there are no segmented fields in the dictionary. The driver option `map fixed length to variable` may be set, but you must then review and edit the **`ora3xdt.sql`** file before creating the repository.
- UNIFACE assigns a special meaning to some characters in Varchar2, Char and Long fields in the Repository. UNIFACE does not work correctly if these characters are corrupted when ORACLE converts character data between character sets.

  Define the ORACLE client character set to avoid this problem. You can do this by using the ORACLE `NLS_LANG` environment variable. See chapter 9 *System parameters* for more information on the

NLS_LANG environment variable, and section 4.8 *ORACLE character sets*, for more information on the configuration of ORACLE character sets.

- To improve performance, it is recommended that (if your system setup can support it) you allow the ORA driver to open between 150 and 250 cursors when accessing the repository in ORACLE. The amount you need depends on how extensively you use the features of UNIFACE Seven. See the description of the `open cursors` driver option in section 4.6.1 *Cursors and statement caching* for more information. When you test forms in UNIFACE Seven, you may want to allow extra cursors for the tables accessed by those forms. This is only necessary if you access the test data on the same logon path as the Repository.

4.2 Paths and logging on

This section describes the following:

- `open` and `close` instructions
- Logon path specification
- Concurrent logon paths
- Logon paths and special services for 3GL
- Driver options affecting logon paths

Connection managed by a transaction manager

If the ORA U5.0 driver is configured and a transaction manager (for example, Encina) is running, the transaction manager controls the connections (rather than the driver). See the documentation for your transaction manager for more information.

4.2.1 open and close instructions

Both of these instructions are supported.

4.2.2 Logon path specification

You can specify the logon path to ORACLE in the following ways:

- In the Proc language open statement:
  open "{*database*}|{*user_name*}|{*password*}", "*path name*"
- Interactively in the DBMS Logon form. The logon form only appears when you have specified a question mark as one or more of the elements of the logon path specification.
- With the following assignment in the assignment file:
  $path_name = ORA:{*database*}|{*user_name*}|{*password*}

The database specification is interpreted by ORACLE, not by UNIFACE. The database can specify one of the following:

- An ORACLE two-task communication driver supplied with ORACLE. For example, the pipe driver on UNIX or the Mailbox driver on Alpha OpenVMS.
- An SQL*Net database specification. See section 3.1 *Networking considerations* for more information on SQL*Net.

Refer to your ORACLE documentation for more information on the syntax of the database specification. ORACLE decides whether to use SQL*Net V1 or SQL*Net V2, depending on the syntax of the database specification. Some examples of complete logon path specifications can be found at the end of this section.

The following are different ways in which you can provide user name and password information:

- Specify both the user name and password.
- Specify the user name, or the password, or both, as a question mark. The UNIFACE logon form appears allowing you to enter the missing information.
- Specify neither user name nor password. In this case, UNIFACE attempts an automatic logon (also known as default logon, or OPS$ logon). This means logging on to ORACLE using the operating system user identification. ORACLE supports automatic logon on some platforms. Whether or not ORACLE supports automatic logon when using SQL*Net depends on the SQL*Net protocol. See your ORACLE documentation for more information.

Specifying a user name without a password, or specifying a password without a user name is not valid. If you do this, the ORA driver will generate the following error:

```
ORACLE Driver Error [-24]: user name without password, or password without user name specified.
```

This error also occurs if you enter an empty user name or password when the DBMS Logon form pops up.

Identifying errors during logon

If you encounter difficulties with the logon path specification, it is recommended that you use the following procedure to identify the source of the problem:

1. Log on to ORACLE using an ORACLE utility available in your environment. This example shows how to log on with SQL*Plus:

   `sqlplus` *user name*|*password@databas*e

   If a problem occurs at this stage, check the ORACLE configuration.
2. Syntactically convert the userid string to a UNIFACE logon path specification in the assignment file, replacing the password by a question mark:

   `$ORA = ORA:`*database*|*user name*|?

   Enter the SQL Workbench and enter a simple SQL query. For example: `SELECT USER FROM DUAL`

   As soon as you DETAIL, the logon form for the $ORA path pops up. Enter the password.

   If a problem occurs at this stage, check the message frame for error messages. Then check the relevant environment variables. For example, `ORACLE_HOME`, `ORACLE_SID` and `TWO_TASK`. See chapter 9 *System parameters*, for more information.

The following examples show logon path specification in the assignment file.

Example 1

```
$ORA = ORA:|scott|tiger
```

This specifies a logon path to the default database, using the default communication driver. The user is `scott` and the password is `tiger`. The mechanism for identifying the default database and default communication driver are specific to the operating system. For example, on UNIX, the environment variable `TWO_TASK`, or the combination of `ORACLE_HOME` and `ORACLE_SID` is used.

Refer to your ORACLE documentation for more information.

Example 2

```
$ORA = ORA:P:|scott|tiger
```

This specifies a logon path to the default database, using the ORACLE two-task pipe driver (UNIX).

Example 3

```
$ORA = ORA:2:|scott|tiger
```

This specifies a logon path to the default database, using the ORACLE Mailbox driver (Alpha OpenVMS).

Example 4

```
$ORA = ORA:t:amsterdam:finance|scott|tiger
```

This specifies a logon path to the `finance` database on TCP/IP host `amsterdam`, using the SQL*Net V1.2 TCP/IP driver. The syntax is confusing because both ORACLE and UNIFACE use the colon as a special character. In this example, the database specification interpreted by ORACLE is `t:amsterdam:finance`.

The equivalent userid string with ORACLE tools for this example is:

```
scott/tiger@t:amsterdam:finance
```

This complicated userid string shows that it is probably not a good idea to expect the end user to enter database specifications in the DBMS Logon form. Define the database specification in the assignment file, or use the ORACLE environment variable `TWO_TASK`. It is possible that on your operating system, ORACLE also supports an alias mechanism to define the database specifications in a configuration file or environment variable (for example, SQL*Net on OS/2 and MS-DOS).

Example 5

```
$ORA = ORA:finance|scott|tiger
```

This specifies a logon path using SQL*Net V2. The database specification `finance` must be defined in an SQL*Net V2 configuration file called `TNSNAMES.ORA`.

Example 6

```
$ORA = ORA:|?|?
```

This assignment causes UNIFACE to display the logon form when the path $ORA is referenced. The end user can specify the user name and password but is not allowed to enter a database specification.

Example 7

```
$ORA = ORA:||
```

This assignment is equivalent to not assigning $ORA at all; it specifies an automatic logon to the default database. The logon form will not display, but UNIFACE attempts an automatic logon to ORACLE when ORACLE is configured to allow automatic logons.

4.2.3 Concurrent logon paths

UNIFACE supports a theoretical maximum of 32,767 concurrent logon paths to ORACLE. See also section 4.4.1 *Transactions* for more information on transaction control when using multiple logon paths.

It is not possible to open multiple concurrent logon paths using the ORACLE single-task driver. The single-task driver is only supported on specific operating systems (for example, OpenVMS), and only when the application is linked single-task with ORACLE. When a logon path is opened with the single-task driver, you can open additional concurrent logon paths using a two-task driver. For example, you can specify the Mailbox two-task driver as the database in the UNIFACE logon path specification.

On some systems, it is possible that ORACLE does not support multiple concurrent logon paths (for example, single-user ORACLE under MS-DOS on a PC).

It is not advisable to open many concurrent logon paths. This will degrade performance, and managing multiple transactions is difficult. If, however, the application must access multiple databases using multiple logon paths, opening concurrent logon paths is a better solution than repeatedly opening and closing a logon path.

If your application requires a second logon path when accessing a global UOBJ, you should consider the following alternatives which require only one logon path:

- Set up synonyms in your schema to access the global UOBJ and arrange for the appropriate privileges to access UOBJ in the central schema.
- Create a local UOBJ in your schema.

These alternatives speed up the initialization of your application. See the UNIFACE online help for more information on UOBJ.

In the same way, UNIFACE Seven does not need to open two logon paths to access the Repositories in an application model.

4.2.4 Logon paths and special services for 3GL

If you are using a U3.*x* driver, the connection for the first logon path to ORACLE is created using the Pro*C precompiler interface. The first logon path is associated with the so-called default connection (created with `EXEC SQL CONNECT` without an `AT` clause).[1] This allows user-defined 3GL to access ORACLE on the same logon path as UNIFACE using the precompiler interface. Accessing ORACLE on the first logon path using the ORACLE Call Interface (OCI), or creating independent connections to ORACLE with either the precompiler interface or the OCI is also supported.

All logon paths except the first logon path are created as independent non-default connections using the ORACLE Call Interface. These logon paths are not accessible by user-defined 3GL.

The 3GL service functions `UGETUOPENFLAG` and `UGETULDA` are provided to access ORACLE logon information. Refer to chapter 13 *Accessing ORACLE from 3GL* and the *3GL Interface Manual* for more information on these functions.

1. The 'first logon path' is either the first logon path to ORACLE, opened since the application was started, or the first logon path to ORACLE, opened after the previous first logon path was closed.

4.2.5 Driver options affecting logon paths

By default, if you are using the U3.*x* driver, UNIFACE creates the first connection to ORACLE using the Pro*C precompiler interface. Although this is completely transparent to most applications, it may cause an incompatibility with user-defined 3GL which creates the precompiler default connection.

If the `disable precompiler connect` driver option is set, UNIFACE only creates independent non-default connections using the OCI, thereby allowing user-defined 3GL to create the default connection with the precompiler interface.

4.3 Retrieving data

This section describes the data retrieval mechanisms supported by ORACLE.

4.3.1 Retrieve profiles

UNIFACE automatically uses the ESCAPE clause with the `LIKE` operator to prevent ORACLE wildcard characters in UNIFACE retrieve profiles from being interpreted. The backslash character (\) is used as the escape character. Some examples of UNIFACE retrieve profiles and their corresponding ORACLE `LIKE` profile are shown in table 4-1:

Table 4-1 How UNIFACE retrieve profiles are mapped to ORACLE retrieve profiles.

UNIFACE retrieve profile	ORACLE LIKE profile
c_c GOLD *	c_c%
c_c GOLD *	c_c%
c% GOLD *c	c\%%c

Retrieve profiles on character string fields behave differently on fields with the Char storage format than on fields with the Varchar2 storage format. For example, when the value UNIFACE is stored in a Char(10) and a Varchar2(10) field, respectively, the value matches the profile GOLD *FACE on the Varchar2 field, but not on the Char field. The reason for this is that the Char field stores trailing spaces, and contains 'UNIFACE ' (three trailing spaces). The profile UNI GOLD * matches on both fields.

As a general rule, retrieve profiles on fixed-length Char fields only give the expected result if the last non-blank character is the GOLD * profile character, or if there is no GOLD * profile character in the retrieve profile.

4.3.2 where clause

The where clause is supported. All text between the double quotation marks ("") is inserted literally as entered. For example, read where "EMPNO > 10" results in the where clause:

```
WHERE (EMPNO > 10) AND (...)
```

Because the text is inserted literally as entered, values must be entered in a format acceptable to ORACLE, taking National Language Support rules into account.

4.3.3 u_where clause

ORACLE handles the u_where clause, but it cannot be applied to Long, Long Raw, BLOB, and CLOB fields in ORACLE.

4.3.4 order by qualifier

The order by qualifier is supported using both ascending and descending order, but cannot be applied to Long, Long Raw, BLOB, and CLOB fields. Sorting is always performed by ORACLE.

4.3.5 selectdb instruction

ORACLE handles this instruction, but it cannot be applied to Long and Long Raw fields in ORACLE.

4.4 U3.*x* transaction control

This section describes the ORA driver's transaction handling when using a U3.*x* or the U4.0 driver.

If you are using the U5.0 driver, transaction management may be performed by a separate transaction manager using the XA Interface. You should consult the documentation for your transaction manager for information on the transaction manager behavior.

4.4.1 Transactions

A transaction consists of all Data Manipulation Language (DML) statements occurring on one logon path (ORACLE session). A transaction ends when a `commit` or `rollback` is performed on the logon path, or when the logon path is closed. If there are multiple concurrent logon paths open to ORACLE, there is one independent transaction on every logon path.

If you are using the U5.0 driver, and a transaction manager is running, the ORA driver does not perform any transaction management, and the rest of this section is not applicable.

ORACLE commits a transaction whenever a schema object is created. This can disrupt the transaction management of your applications. It is therefore, recommended that you use the Create Table utility, instead of creating objects on the fly.

If your application must access data in multiple ORACLE databases, you can achieve this in one of the following ways:

- Configure the ORACLE databases in a distributed database. This requires SQL*Net and the optional Distributed Database Extension. You can then access the distributed database on one logon path by setting up synonyms in one of the databases to allow UNIFACE to access objects in the other databases. In this case, the ORACLE

distributed database ensures that all DML operations are in one transaction.

- Access the ORACLE databases on two or more logon paths, using either SQL*Net or PolyServer. In this case, DML operations on different logon paths are not in one transaction. There is one independent transaction on every logon path.

Similarly, if your application has to access data in multiple schemas in the same database, you can achieve this in one of the following ways:

- Access all schemas on one logon path by setting up synonyms in one of the schemas to allow UNIFACE to access objects in the other schemas. In this case, ORACLE ensures that all DML operations are in one transaction.
- Access the ORACLE database on two or more logon paths. In this case, DML operations on different logon paths are not in one transaction. There is one independent transaction on every logon path.

4.4.2 Locking

Types of locking supported

All UNIFACE locking types are supported: optimistic, cautious, and paranoid. Refer to the *UNIFACE Reference Manual* for an explanation of these types.

UNIFACE locks individual rows only. It does this by using the `SELECT ... FOR UPDATE NOWAIT` SQL statement.

Rows are unlocked when the `commit` or `rollback` instructions are executed, when the application ends, or when the logon path to ORACLE is closed.

Default locking type

The default locking type is cautious. You can override cautious locking for individual entities by specifying optimistic locking in the Entity definition form.

read/lock instruction

You can specify paranoid locking by using the `read/lock` Proc instruction in the Read trigger of an entity.

U_VERSION

The ORA driver supports the `U_VERSION` mechanism to improve locking performance. Stored packages created by UNIFACE support requests for optimistic locking with `U_VERSION`.

4.4.3 commit and rollback

Both of these instructions are supported by ORACLE. There are no special considerations.

Two-phase commit

This is supported by ORACLE if the optional Distributed Database Extension is used. Because ORACLE itself uses two-phase commit, the UNIFACE two-phase commit mechanism is not used; you do not need to use the setting `$TWO_PHASE_COMMIT`. Two-phase commit is completely transparent to UNIFACE and is used automatically by ORACLE when required. All DML statements which occur on one logon path to an ORACLE distributed database are in one transaction.

4.5 Implementation of segmented interfaces

This section provides some background information on the implementation of the segmented field interface with ORACLE (BLOB support). You may find this information useful when faced with the alternatives of using UNIFACE proprietary variable-length techniques (overflow tables), or using the segmented field interface. The U4.0 and U5.0 drivers implement the segmented field interface in a different way to the U3.x drivers. The behavior of both drivers is described below.

UNIFACE uses segmented I/O provided by ORACLE when reading a field with a segmented interface. There are no special considerations.

The U4.0 and U5.0 drivers implement BLOBs as CLOB and BLOB data types and which are also written to the database in segments. These data types were introduced in ORACLE8.

The U3.*x* drivers implement BLOBs as LONG and LONG RAW data types. However, for this data type, ORACLE does not provide segmented write capabilities. To write data into a Long or Long Raw field using the segmented interface, UNIFACE must allocate a contiguous storage area as large as the field to be written using operating-system dynamic memory allocation. This is subject to platform-specific memory limitations. The memory is allocated on the client platform when SQL*Net is used, and on the server platform when PolyServer is used. The memory allocation occurs when the segmented field is being written, and memory is released immediately afterwards.

Caution: If you use segmented fields on a platform more restrictive than a 32-bit virtual memory operating system, you can encounter severe restrictions when attempting to update or insert a record. Furthermore, the update or insert of large segmented fields can cause other activities on your system to fail. You should consider using the UNIFACE proprietary variable-length techniques instead of the segmented field interface.

If the amount of memory required to write the segmented field cannot be allocated, the following errors can occur:

```
ORACLE Driver Error[-75]: Actual length of LONG or LONG RAW data exceeds
platform specific memory limitation.

ORACLE Driver Error[-4]:  Dynamic memory allocation failed.
```

There is no solution to the first error. However, if the second error occurs, increasing available memory can solve the problem (for example, increase storage available for paging and swapping with virtual memory operating systems).

If the limitation described above is unacceptable, consider using UNIFACE proprietary variable-length techniques (overflow tables), instead of the segmented interface. Set the `disable segmented field` driver option to achieve this. See section 8.2 *Modify packing code mapping* for more information.

4.6 Performance issues

This section describes how to improve the performance of your ORACLE database with UNIFACE. Read this section in combination with the performance-tuning section in your ORACLE documentation.

4.6.1 Cursors and statement caching

This section describes the statement cache which is managed by UNIFACE, and provides some guidelines for setting the `open cursors` driver option.

Statement cache

For each statement issued to ORACLE, the U3.*x* drivers open a cursor while the U4.0 and U5.0 drivers open a statement handle. Both terms refer to placeholders. In both cases, the action is managed by UNIFACE in a statement cache, as described below. This section uses the generic term, *statement marker*, to refer to a cursor (in U3.*x*) or statement handle (in U4.0 and U5.0).

UNIFACE manages a fully associative Least Recently Used (LRU) cache of SQL and PL/SQL statements. When a statement occurs for the first time, SQL or PL/SQL is generated, parsed on an ORACLE statement marker, and placeholders are bound. The statement marker remains open. When the same statement recurs, there is no need to generate and parse (or prepare) it again, and in many circumstances placeholder rebinding is not required. The statement marker is just reexecuted. This cache is located in the ORA driver, and it is completely independent from the cache of shared SQL areas in the ORACLE server. UNIFACE manages one independent statement cache per logon path to ORACLE. The statement markers in a cache are closed when the logon path to ORACLE is closed.

If a statement is not yet in the cache, and the maximum number of open statement markers allowed by the `open cursors` driver option has not been reached, UNIFACE opens a new statement marker for that statement. If the maximum number of open statement markers has been reached, the statement marker which has been used least recently is chosen for reparsing. The statement which was associated with this statement marker is removed from the cache. The new statement is generated and parsed on the selected statement marker.

Note: The purpose of the statement cache managed by UNIFACE is to reduce the number of parse calls. The purpose of the cache of shared SQL areas managed by the ORACLE server is to reduce the number of parse operations. See your ORACLE documentation for an explanation of the difference between parse calls and parse operations.

The statement cache improves performance in the following ways:

- The cache reduces the number of parse calls.
- The cache reduces the client/server communication overhead, which is particularly important when using SQL*Net.

The effectiveness of the cache is further improved when UNIFACE uses stored packages, as only one procedure call needs to be cached for the implementation of multiple I/O requests.

The UNIFACE statement cache mechanism is not used for statements issued in the SQL Workbench and the `sql` Proc instruction.

SQL and PL/SQL statements generated by the driver use placeholders rather than actual data in the statement. This improves the reusability of the statement, as there is no need for reparsing when the only thing that changes is the data. UNIFACE limits the number of statements in the cache which have a user-defined `where` clause (the Proc `read where` clause), as such statements can include actual data, and the majority of them will not be reexecuted.

In exceptional cases, UNIFACE has to reparse an SQL query. If the number of rows fetched is less than the number requested by UNIFACE, and at the same time UNIFACE is using all of the ORACLE array fetching, deferred parsing and combined `execute/fetch` features, ORACLE considers the parsed statement not valid. UNIFACE must then reparse it the next time it needs to be executed. UNIFACE avoids this problem in some cases by using heuristic methods to estimate the number of rows to be fetched—it is then possible to use nondeferred parsing.

You can observe the behavior of the cache to a limited extent by monitoring SQL in the message frame (with the `/pri=32` command line switch). SQL or PL/SQL statements only appear in the message frame when they are generated. When a statement in the cache is reexecuted, no text is added to the message frame. If you want to observe the LRU caching behavior, you should set the `open cursors` driver option to a very low number.

Alternatively, you can obtain statistics on the effectiveness of the statement cache by using ORACLE's tracing facilities. Refer to your ORACLE documentation for more information.

Driver options affecting the use of cursors

The driver option `open cursors` specifies the maximum number of statement markers or statements per logon path that UNIFACE is allowed to open. The default value is 45. The absolute minimum value is four.

There is no absolute maximum value, as far as UNIFACE is concerned. The maximum value is an initialization parameter of the ORACLE server to which the application connects. When the `open cursors` value in the `USYS$ORA_PARAMS` assignment exceeds the maximum number of statement markers allowed by the ORACLE server, ORACLE can generate an error, but this only happens once UNIFACE has actually opened the maximum number of statement markers.

Recursive statement markers might also be opened by ORACLE on behalf of the application. (See your ORACLE documentation on 'recursive calls' and 'recursive cursors' for more information.) UNIFACE cannot, therefore, guarantee that the maximum number of statement markers allowed by the ORACLE server is not exceeded:

```
ORA-1000  Maximum open cursors exceeded
```

This error is most likely to occur if the UNIFACE statement cache has already opened its maximum number of statement markers, and a table is created on the fly. It is recommended that you do the following to avoid this problem:

- Create tables, indexes, and packages using the Create Table utility instead of creating these objects on the fly.
- Configure the ORACLE server to allow some extra statement markers, in addition to the maximum number of open statement markers you allow UNIFACE to open. In this way, you give yourself a safety margin.

You must also take into account statement markers opened by:

- Database triggers which you create
- User-defined 3GL accessing ORACLE on the same logon path as the driver
- PL/SQL called in the `sql` Proc instruction and in the SQL Workbench

You must reserve extra statement markers for these purposes as the ORA driver only counts the statement markers that it opens. Reserve a 'safety margin' of extra statement markers by configuring the ORACLE server to allow more statement markers than the ORA driver is allowed to open.

You can configure the maximum number of statement markers per session allowed by the ORACLE server by setting the ORACLE initialization parameter `OPEN_CURSORS` (a parameter in `INIT.ORA`). You must shut down and restart the server after changing its value. The default value of the `open cursor` driver option is 45, which is less than the default value 50 of the `OPEN_CURSORS` parameter in ORACLE. This already provides a small safety margin.

The value of the `open cursors` driver option can be used to tune private SQL and PL/SQL areas and to reduce the number of parse calls issued by your UNIFACE application.

Calculating the number of cursors needed

It is recommended that you begin with the following guidelines to compute an appropriate value for the `open cursors` driver option. For every table accessed by an application on one particular logon path, add the number of cursors required, as shown in table 4-2:

Table 4-2 Number of cursors opened when you access a UNIFACE table.

Table access method	UNIFACE base table	UNIFACE overflow table
Stored package	6	Not applicable
Dynamic SQL (read only)	8	2
Dynamic SQL (read/write)	10	4

You may want to add additional statement markers, based on the knowledge of how a particular table is accessed by your applications. For example, the use of `read u_where` and `selectdb` Proc instructions or `order by` clauses, and the use of various retrieve profiles can require extra statement markers.

Determine the total number of statement markers per logon path for your application. Take the maximum of these values and set the `open cursors` driver option to that value. Do this for all your applications. Different applications may use different values for the `open cursors` option, which means that you will have to arrange for your

applications to read different assignment files. Ensure that the ORACLE OPEN_CURSORS initialization parameter is at least equal to the largest value of open cursors set for the applications connecting to the server. As already described, allow a safety margin of some extra statement markers to avoid problems with recursive statement markers opened by the ORACLE server.

The following examples show how to calculate the number of statement markers required by an application. The following three tables are defined:

- Table EMP is a table which has an associated stored package.
- Table DEPT uses UNIFACE variable-length techniques (overflow table).
- Table PROJECTS uses UNIFACE variable-length techniques (overflow table).

Application HIRE uses two concurrent logon paths, $ORA1 and $ORA2. It accesses the PROJECTS table via the $ORA1 path to database personnel2. Ten statement markers are required for the PROJECTS base table, and four statement markers for the PROJECTS overflow table. It accesses the EMP table via the $ORA2 path to database personnel1. Six statement markers are required for the EMP table.

The total number of statement markers required, therefore, is 14 for the $ORA1 logon path, and six for the $ORA2 logon path. Choosing the maximum of the values, therefore, means that the open cursors driver option should be set to a value of 14.

A second application FIRE uses one logon path $ORA to the personnel1 database. Stored packages are disabled. Six statement markers are required for the EMP table. The DEPT table is accessed read only, which means that eight statement markers are required for the base table and two are required for the overflow table (see table 4-2 for information on the number of statement markers required when accessing a table). In this case, the open cursors driver option should be set to a value of 16.

In this example, the server for database personnel1 should allow at least 16 open statement markers, and the server for database personnel2 should allow at least 14 (not six statement markers as computed for logon path $ORA2, but the maximum value of open cursors for all applications connecting to it).

Note: This calculation does not yet include the extra cursors which you should reserve for ORACLE recursive cursors, database triggers, user-defined 3GL and PL/SQL called in the sql Proc instruction or in the SQL Workbench.

The numbers which are presented as guidelines in table 4-2 are given in the assumption that you want to improve performance and reduce client/server communication overhead, at the cost of extra private SQL areas allocated by the ORACLE server. This is usually a good choice in client/server environments (especially when using SQL*Net).

If, however, you are accessing ORACLE locally on a platform with memory limitations, you may want to allow UNIFACE fewer open statement markers than suggested by the guidelines above. Do not restrict the number of statement markers to less than two per open table on any particular logon path. If you do specify less than this, the following error can occur:

```
ORACLE Driver Error[-81]: No more cursors available for statement processing.
Increase value of 'open cursors' in USYS$ORA_PARAMS.
```

You can find more information on tuning ORACLE memory allocation in your ORACLE documentation which also provides information on how to trace an application and observe its behavior with respect to parse calls. It is a good idea to perform this tracing in a production environment, as the figures described in this section are only guidelines.

4.6.2 Stepped hitlist

The default step size is dependent on the operating system; it is `10` on most operating systems, but the number may be different (not necessarily smaller) on platforms with memory limitations.

When UNIFACE is building the stepped hitlist, it uses ORACLE array fetching to reduce the client/server communication overhead. This does, however, mean extra memory usage by the driver.

If you want to ensure that the step size is a specific fixed value, do not depend on the default value. Instead, use one of the following driver options:

- `array fetch size`
- `fixed array size`
- `step size`

array fetch size

This option specifies the minimum number of records UNIFACE fetches with one call to the ORACLE server using array fetching. The number specified must be in the range of 1 through 32,767.

This option indirectly specifies the amount of memory which is dynamically allocated by the ORA driver. The driver allocates an I/O buffer of approximately 15 kilobytes multiplied by the array size plus one. When you are using SQL*Net, memory is allocated on the client platform. When PolyServer is used, memory allocation occurs on the PolyServer platform. The I/O buffer is allocated when first needed, and the memory is released when the number of logon paths to ORACLE drops to zero.

The specified array size is subject to platform-specific memory limitations. When the amount of memory required to fetch an array of the specified size cannot be allocated, the following errors can occur:

```
ORACLE Driver Error[-54]: Storage required for I/O buffer exceeds system limits.
Reduce array size.
```

and:

```
ORACLE Driver Error[-4]:  Dynamic memory allocation failed.
```

If you encounter driver error -54, reducing the array size solves the problem. If driver error -4 occurs, both reducing the array size and increasing available memory can solve the problem (for example, increase the storage available for paging and swapping with virtual memory operating systems).

fixed array size

By default, UNIFACE and the ORA U3.*x* driver make optimal use of the memory that is allocated for ORACLE array fetching. The size of the allocated memory is calculated based on the size of the largest possible record. When smaller records are fetched, the array fetch size is automatically increased to reduce the client/server communication overhead as much as possible. The step size of the UNIFACE stepped hitlist must be a multiple of the array fetch size which has been computed in this way. The step size may, therefore, be very large. Because the array size depends on the size of the record, the step size may vary for different tables.

If you want to avoid having large and varying step sizes, you can set the `fixed array size` option. The `fixed array size` option forces UNIFACE to use exactly the array size specified with the `array fetch size` option (or its default value), instead of dynamically computing the maximum array size.

step size

This option specifies the step size of the UNIFACE stepped hitlist mechanism. The number specified must be in the range from 0 through 32,767. If you specify 0, the stepped hitlist mechanism is disabled, causing UNIFACE to internally finish the hitlist each time, before returning control to the application.

The actual step size used is always either zero or a multiple of the actual array fetch size. When necessary, UNIFACE uses the smallest step size greater than or equal to the specified step size which is a multiple of the array size. To avoid confusion, specify a step size which is either zero or a multiple of the specified array size.

When the `fixed array size` option is set, and the recommended step size is a multiple of the array fetch size, the `array fetch size` option only affects the performance of the application, and not the behavior. The `step size` option, on the other hand, determines how many records are processed before control returns to the application. This affects, for example, the behavior of the `$currhits` variable.

Example:

```
USYS$ORA_PARAMS = array fetch size 5, step size 10, fixed array
size
```

If you use the assignment in this example, UNIFACE receives five records at a time from the ORACLE server, and internally processes the records in steps of 10 (two arrays of five records are fetched per step of the hitlist). This is backwards compatible with older versions of the ORA driver, which used a fixed step of 10. The performance is better because array fetching is used.

4.7 Extended driver behavior

Performance tuning read option

The Proc `read` instruction has been enhanced with an `option` flag. The `option` parameter allows you to adjust various DBMS performance-related parameters, such as the maximum number of returned hits, the step size of a query, and the hit cache size.

For more information on the use of the `read` Proc statement, see the *Proc Language Reference Manual*.

Field subsetting

Fields which are not painted on the UNIFACE form are longer selected, feteched, inserted, or updated by the DBMS driver. The UNIFACE kernel determines which fields are not required.

4.8 ORACLE character sets

UNIFACE supports the use of multibyte character sets in combination with ORACLE NLS multibyte character sets.

The character set of an ORACLE database is established when a database is created. ORACLE automatically converts character data (Char, Varchar2, Varchar, and Long storage formats) between the database character set and the character set requested by the client (for example, if another character set is specified in the `NLS_LANG` environment variable).

When storing data in ORACLE with UNIFACE, character sets are usually converted transparently, but this is not the case when you run the ORA driver in a 8-bit environment which connects to an ORACLE server in a 7-bit environment. For example, when the ORA driver runs in an environment where the default character set is WE8DEC (8-bit DEC West-European) and it connects to an ORACLE server installed with US7ASCII (7-bit ASCII), some characters which are meaningful to UNIFACE are corrupted when stored and retrieved. This problem is solved by specifying the WE8DEC character set in the `NLS_LANG` environment variable (set `NLS_LANG` to american_america.we8dec). For more information on ORACLE environment variables, see chapter 9 *System parameters*.

4.9 Assignment settings

You can use the $E6_BASEDATE assignment setting to store 0-1-0000 for empty Time and DateTime fields. The format is as follows:

```
[SETTINGS]
$E6_BASEDATE 0000
```

This assignment setting is *only* available for ORACLE.

See *Migration to UNIFACE V7.2* for important information about migrating E6 packing data from earlier versions of UNIFACE.

Chapter 5 sql and the SQL Workbench

Both the `sql` Proc instruction and the SQL Workbench are supported.

If a transaction manager is running, commit and rollback is managed by the transaction manager rather than the ORA driver. When you issue the `commit` and `rollback` Proc instructions, they are not passed to the ORA driver but to the transaction manager driver. The `commit` and `rollback` instructions should not be used in the SQL workbench. For further information, see the documentation for your transaction manager.

All data which is selected, except for Long Raw or BLOB fields, is converted to the Varchar2 storage format by ORACLE with a maximum length of 2000 characters.

Long Raw or BLOB data is converted by UNIFACE to a character string in hexadecimal format, with a maximum length of 2000 characters.

If data is truncated when it is returned to UNIFACE, no error is generated.

In the SQL Workbench, the maximum length of one row of the formatted result is 8190 bytes. When this length is exceeded, the ORA driver generates the following error:

```
ORACLE Driver Error [-27]: Selected data too large for SQL Workbench.
```

Data formats in the `sql` Proc instruction and SQL Workbench are dependent on ORACLE National Language Support parameters. This is the case both for input and output data.

5.1 Special considerations for SQL

5.1.1 $result and $status

When you use the `sql` Proc instruction, the driver returns values to `$result` and `$status` that depend on the SQL statement used. The following list describes a few different circumstances:

- If the SQL statement used in the `sql` Proc instruction is a `SELECT` statement which succeeds in selecting one or more rows, the first field of the last row is returned as a character string in `$result`. The conversion to character string occurs as described above. The number of rows selected is returned in `$status`.[1]
- If the SQL statement is not a `SELECT` statement, or if the `SELECT` statement succeeds but selects zero rows, `$status` has a value of zero and `$result` is not modified.
- If the SQL statement fails, `$status` has a value less than zero and the value of `$result` is undefined.

In the SQL Workbench, the return status is shown in the message area.

5.1.2 Other considerations

In the SQL Workbench, UNIFACE presents data using a fixed-column length format. To preserve this format, UNIFACE rearranges the order of the selected columns by moving all Long, Long Raw, CLOB, and BLOB fields to the end. In the `sql` Proc instruction, the first field returned in `$result` is always the first field selected in the query, even if it is a Long or Long Raw field.

SQL statements must *not* be terminated with a semicolon and you *cannot* use embedded SQL.

The ORA U3.*x* drivers supplied with UNIFACE Seven retrieve all data when using a `select` statement in the SQL Workbench using Q call mode 2. The ORA U2.0 driver for UNIFACE Seven and UNIFACE Version 5 including the U3.*x* drivers for UNIFACE Version 5 use Q call mode 2 with subsequent Q calls of mode 3.

1. Do not use the `sql` Proc instruction to count rows in this way, as this is very inefficient. If you want to count rows, use the `selectdb` or `lookup` Proc instructions.

Note: If you are using ODBC to manipulate numeric data, and the data returned from ORACLE contains more than 32 digits, the ORA driver truncates the returned data to 32 digits.

If you are using the U4.0 or U5.0 driver and wish to use rowid in a query, you must explicitly convert the rowid to or from character format to the ORACLE8 internal rowid format. Thus, `SELECT` statements should select `ROWIDTOCHAR(ROWID)` rather than `ROWID`, and conditions should have the syntax '`ROWID=CHARTOROWID(`'*rowid_string*'`)`'.

5.2 Special considerations for PL/SQL

PL/SQL is supported in the `sql` Proc instruction and SQL Workbench, provided that the ORACLE server to which you connect has the Procedural option. Calling stored procedures and functions is possible. PL/SQL statements and statement blocks are terminated by a semicolon.

PL/SQL code must be included in an anonymous PL/SQL block. For example:

```
sql "begin raise_application_error(-20000, 'I goofed!'); end;", "ora"
```

When this statement is executed, `$status` returns a negative value and `$dberror` contains the error number which is generated. ORACLE errors in `$dberror` are always positive; `$dberror` returns `20000` in this case.

As a special service for PL/SQL, UNIFACE allows you to pass one data item back to UNIFACE, provided that this data item is convertible to the Varchar2 storage format and has a maximum length of 2000 characters. To achieve this, UNIFACE binds a placeholder called `:uresult` when it is referenced in the PL/SQL block. (See your ORACLE PL/SQL documentation for more information on placeholders.)

For example:

```
sql "begin select user into :uresult from dual; end;", "ora"
```

The value which is stored in `:uresult` by the PL/SQL block is available as a character string in the Proc variable `$result`. The conversion to a character string occurs as described above. When the PL/SQL block reads the value of `:uresult`, it is a Varchar2 (2000) variable which holds the NULL value.

The situations which can arise, and the values they return, are as follows:

- If a PL/SQL statement which does not reference the `:uresult` placeholder succeeds, `$status` has a value of zero and `$result` is not modified.
- If a PL/SQL statement references the `:uresult` placeholder, but either assigns no value or the NULL value to it, `$status` has a value of `1` and `$result` is the empty string.
- If a PL/SQL statement succeeds and assigns a non-NULL value to the `:uresult` placeholder, `$status` has a value of `1` and `$result` contains the value assigned to the `:uresult` placeholder converted to a character string.
- If the PL/SQL statement fails then `$status` returns a value less than zero and the value of `$result` is undefined.

In the SQL Workbench, the return status is shown in the message area.

As described for the `sql` Proc instruction, the SQL Workbench also supports PL/SQL and the `:uresult` placeholder. This means that you can test PL/SQL blocks in the SQL Workbench.

5.3 Monitoring SQL in the message frame

When UNIFACE logs on to ORACLE, the driver displays a message which identifies the driver version, the driver options in the USYS$ORA_PARAMS assignment and the current major and minor package version numbers of the driver (see section 6.2 *Stored package version*, for more information on package versions). For example, the following message appears:

```
U3.3 ORA driver for ORACLE 7.3.
Driver options: step size 20.
The current major package version of the driver is: 1.
The current minor package version of the driver is: 0.
```

Apart from this message, only SQL and PL/SQL issued by UNIFACE is displayed in the message frame.

Because of the way UNIFACE uses SQL and PL/SQL with ORACLE, the information you can see in the message is restricted in the following ways:

- UNIFACE displays only those statements which are both parsed *and* executed. For example, the table existence check which is performed when a table is accessed for the first time is not shown, because it is only parsed and not executed.
- UNIFACE uses a statement cache, which means that it reexecutes statements which have already been generated and parsed. When this occurs, no SQL is regenerated, and therefore no SQL appears in the message frame.
- UNIFACE uses placeholders and host language binding, instead of supplying actual data in the SQL statement. This means that you cannot see data in the message frame. Placeholders are identifiers, preceded by a colon. The placeholder names are generated by the ORA driver, and are generally not very meaningful. For example:

  ```
  SELECT "USER" FROM "DUAL" WHERE "DUMMY" = :"WPH1"
  ```

- UNIFACE implements `commit` and `rollback` using functions in the ORACLE Call Interface, instead of using SQL statements. You cannot, therefore, see the `commit` and `rollback` statements in the message frame.
- UNIFACE implements multiple I/O requests by means of one stored procedure. You can see the procedure call as an anonymous PL/SQL block in the message frame, but you cannot see which I/O request is being called. When multiple I/O requests occur, you usually see the procedure call only once, as it is cached in the statement cache managed by the ORA driver.

Chapter 6 Stored packages

UNIFACE generates stored packages containing procedures for most basic I/O operations.

Caution: UNIFACE generates stored packages for only one reason—to improve performance. Do not make changes to the packages generated by UNIFACE. Do not make any assumptions about the way that UNIFACE uses the stored packages. For example, in situations where you might expect UNIFACE to call a procedure in a stored package, it uses dynamic SQL instead.

Using stored packages provides better performance than using the equivalent dynamic SQL statements, for the following reasons:

- The stored package is parsed and compiled only once, then stored in the database. Dynamic SQL statements, however, are parsed and compiled *every* time they are issued by an application unless they are present in the cache of shared SQL areas managed by the ORACLE server.
- UNIFACE includes multiple I/O requests in one procedure per table. This means that UNIFACE executes multiple statements with the client/server communication overhead of only one statement. That is the same amount of overhead required by only one dynamic SQL statement. The economy afforded by stored packages is especially important when using SQL*Net.

UNIFACE can only create and use stored packages at run time if the ORACLE server to which the application connects has the Procedural option. In UNIFACE, you can always generate SQL scripts to create the packages. Do this with Deployment–>Database Utilities–>Create Table.

UNIFACE associates one stored package with a table in ORACLE. For each package, it creates both a package specification and a package body. The name of the package is the name of the associated table, with $U appended. Tables and packages are in the same name space in ORACLE. The presence of the dollar sign ($) in the package name normally ensures that there are no naming conflicts, as the $ is not allowed in UNIFACE entity names.

Tables which meet any of the following conditions do not have an associated package:

- The table contains a Long or Long Raw column.
- The table is defined with 'No updates' in the application model.
- The table name is one of the following reserved words:

  ```
  CHECK_EXISTENCE_OF_PRIMARY_KEY
  DELETE_ROW_BY_PK_AND_UVERSION
  DELETE_ROW_BY_PRIMARY_KEY
  DELETE_ROW_BY_ROWID
  DUMMY_FIELD
  LOCK_ROW_BY_PRIMARY_KEY
  LOCK_ROW_BY_ROWID
  MAJOR_VERSION
  MINOR_VERSION
  NUM_FALSE
  NUM_TRUE
  ONE_ROW_AFFECTED
  SELECT_ROWIDROW_BY_PRIMARY_KEY
  SELECT_ROW_BY_PRIMARY_KEY
  SELECT_ROW_BY_ROWID
  UNIFACE_IO_REQUEST
  UPDATE_ROW_BY_PK_AND_UVERSION
  UPDATE_ROW_BY_PRIMARY_KEY
  UPDATE_ROW_BY_ROWID
  WU_VERSION
  XROWID
  ```

- The table name is identical to an X followed by the name of one of the columns in the table, or the table name is identical to a W followed by the name of one of the columns in the primary key of the table.

If a table meets one of the above conditions, UNIFACE does not create an associated package specification and package body, but instead automatically uses dynamic SQL with that table. You do not need to set the driver option `ignore missing packages` when such a table is accessed.

6.1 Creating stored packages

It is recommended that you use the Create Table utility to generate an SQL script which creates all tables and associated package specifications and package bodies.

If you do not use the Create Table utility, and tables are created on the fly, UNIFACE also creates the associated package specifications and package bodies at the same time. Creating packages on the fly, however, may require very long SQL statements, which can exceed the size of the internal SQL buffer used by UNIFACE (approximately 10 kilobytes). If this happens, the driver generates the following error:

```
ORACLE driver error[-51]: Generated SQL statement too large for internal buffer.
```

To avoid this, use the Create Table utility, instead of attempting to create packages on the fly.

Another error which may occur if you create a package specification and a package body on the fly is:

```
ORACLE driver error[-47]: Package created with compilation errors. Query the
ORACLE data dictionary view USER_ERRORS or DBA_ERRORS for the error messages.
```

When this error occurs, the actual compilation errors are not shown in the message frame. To see these, you must query the ORACLE data dictionary view USER_ERRORS or DBA_ERRORS. You can do this by using the SQL Workbench or the ORACLE utility SQL*Plus. See your ORACLE documentation for further information.

If you attempt to create a package generated by UNIFACE, but the associated table does not exist, obscure PL/SQL compilation errors are generated. This happens because the package uses the PL/SQL %TYPE attribute to declare procedure parameters. This is only valid when the associated table already exists. You will not normally encounter this problem, as UNIFACE first creates the table and then the package.

6.2 Stored package version

UNIFACE assigns every package a major and a minor version number when the package is created, or when the SQL script to create the package is generated. The version numbers are hard-coded in a special procedure, called PACKAGE_VERSION, so that UNIFACE can determine the version at run time.

These version numbers serve to maintain compatibility when the body of the procedure is enhanced in subsequent releases of the ORA driver, and to prevent future releases of the ORA driver from executing older and incompatible versions of the package.

The ORA driver has a current major package version number and current minor package version number. These version numbers must not be confused with the driver version number (for example, U3.3) or with any other version numbers of UNIFACE products. When UNIFACE creates a stored package, or generates SQL scripts in the Create Table utility, the package is assigned the current major and minor version number from the ORA driver.

6.2.1 Compatibility of package versions

When the driver is requested to open a table, and it finds that the table already exists, it calls the `PACKAGE_VERSION` procedure in the associated package to determine the major and minor package version number. When the major version number of the package in the database is not equal to the current major package version number of the ORA driver, the driver generates the following error (the names and version numbers used are only examples):

```
ORACLE driver error[-83]: Major version number of package in the database not
acceptable: Package DEPTP, Major version of package in database is 2, current
major package version of ORACLE driver is 3.
```

This error indicates that an enhancement has been implemented in the ORA driver (this resulted in the major package version of the driver being changed) after the stored package was created, and that the stored package has not been upgraded. If this error occurs, you must upgrade the package as described in section 6.5 *Upgrading the stored packages*.

If the major version is correct, but the minor version number is not equal to the current minor package version, UNIFACE automatically adapts its behavior to use the package in the database. This means that, for example, a performance enhancement is not used.

You can see what the current major and minor package versions of the ORA driver are by running your application with SQL information in the message frame (the `/pri=32` command line switch). The version numbers are then displayed when the driver logs on to ORACLE. For example:

```
The current major package version of the driver is: 3.
The current minor package version number of the driver is: 1.
```

Release notes accompanying a new version or maintenance release of UNIFACE or the ORA driver may recommend or instruct you to upgrade the packages to the current major and minor version of the ORA driver. If you do not upgrade the packages, you may encounter the ORA driver error -83 as previously shown. You can upgrade the packages using the upgrade procedure described in section 6.5 *Upgrading the stored packages*.

6.2.2 Packages used by multiple drivers

If multiple versions of the ORA driver are accessing the same schema, they can only use the packages when all versions of the driver have the same major package version number. If they already have the same major package version number, it is recommended that you upgrade the packages to the highest minor package number used by the drivers. To do this, use the ORA driver with that minor package version number for the upgrade procedure.

If the different ORA drivers do not have the same major version numbers, it is recommended that you install a new maintenance release of UNIFACE or the ORA driver to replace the driver or drivers with the lower major package version number or numbers. Alternatively, you can set the driver option `disable packages` in the `USYS$ORA_PARAMS` assignment, for all or some of the drivers, so that only drivers with identical major package versions are using the packages in the schema.

6.3 Dependency of packages

The definition of the package specification and package body are based on the definition of the associated entity in the UNIFACE application model. If you change the application model entity definitions after the package has been created, or after the generation of the SQL script to create that package, you may have to re-create that package.

Note: Keeping track of the dependencies between packages and the application model can be difficult if the application model changes frequently. Therefore, it is recommended that you set the `disable packages` *driver option during the early stages of development. If you have doubts about the validity of a stored package, follow the upgrade procedure as described in section 6.5 Upgrading the stored packages.*

If you make any of the following changes to your application model, you must create or replace the packages associated with the entities concerned:

- Remove or add an entity, or change its name.
- Change an entity from 'No updates' to modifiable.
- Add or remove a column, or change its name.
- Add a column, or remove a column from the primary key.
- Change the data type or packing code of a column.
- Change the order of fields in the entity or in the primary key.

Caution: If a stored package is not consistent with the definition of the associated table in the application model, very obscure errors can occur at run time.

6.4 Dependency of packages on driver options

Normally, you would not create or upgrade packages because of changes in driver option settings. Exceptions to this rule occur in the following situations:

- If packages were never created, and a driver option which disabled stored packages was set, and that option is reset, you must create packages.
- If packages were created with an older version of the ORA driver, which had an incompatible major package version, and a driver option which disabled stored packages was set, you must upgrade all packages before resetting the driver option.

6.5 Upgrading the stored packages

New UNIFACE or ORA driver releases may involve changes in driver functionality which require that all packages be updated. A package upgrade procedure is provided which you can use for the following purposes:

- To add packages to tables for which the packages were not yet created.
- To upgrade all packages to the current major and minor package version of the ORA driver.
- To update packages to reflect changes in the application model.

Note: If you have any doubts about whether all packages are up-to-date or not, it is always good practice to follow the package upgrade procedure.

6.5.1 The upgrade procedure

The package upgrade procedure is as follows:

1. Set the driver option `upgrade packages` in the `USYS$ORA_PARAMS` assignment. Make sure that the options `disable checks` and `disable packages` are not set.
2. Run the Create Table utility (Deployment–>Database Utilities–>Create Table) for the tables associated with the packages

that you want to upgrade. Because the upgrade packages option is set, the Create Table utility only generates the statements required to upgrade the packages, and does not generate statements to create tables and indexes.

3. Execute the generated SQL script using the ORACLE utility SQL*Plus or SQL*DBA. It is only possible to create the packages in a schema which already contains the associated tables. If you want to create both the tables and the packages, you should be using the normal functionality of the Create Table utility and not this upgrade procedure.
4. Test your applications with the new packages.
5. Remove the upgrade packages option from the USYS$ORA_PARAMS assignment. At this stage, you may set the disable checks option if you want.

When performing the upgrade procedure, you do not need to drop the old packages, because UNIFACE generates the statement:

```
CREATE OR REPLACE PACKAGE . . .
```

However, when you change the name of an entity, or remove an entity or change the application model in such a way that an entity no longer has an associated package, the upgrade procedure does not drop the packages which are no longer used. This is harmless, but the old packages waste space in the database. In this case, it is advisable to drop all packages in a schema before performing the upgrade procedure.

6.6 Driver options affecting stored packages

If you set any of the following options in the USYS$ORA_PARAMS assignment, the use of stored packages is disabled:

- disable packages
- u2 default mapping
- u2 enhanced_mapping
- u2 enhanced_mapping_2

If the use of stored packages is disabled, the behavior of UNIFACE changes as follows:

- UNIFACE does not generate CREATE OR REPLACE PACKAGE and CREATE OR REPLACE PACKAGE BODY statements when you use the Create Table utility.
- UNIFACE does not create package specifications and package bodies when it creates tables on the fly.
- UNIFACE does not attempt to call procedures, but uses only dynamic SQL, even when the packages exist in the database.
- The ORA driver does not display the current major and minor package version in the message frame when the application is monitoring SQL.

You must disable the use of stored packages when connecting to an ORACLE server which does not have the Procedural option. You may want to disable the use of stored packages when working in a development environment where the application model changes frequently, thus causing extra overhead for recreating the packages which are dependent on that application model.

The rest of the options described in this section are applicable only if you have not disabled the use of stored packages.

upgrade packages

If the driver option upgrade packages is set, the Create Table utility generates only the CREATE OR REPLACE PACKAGE and CREATE OR REPLACE PACKAGE BODY statements, and *not* the CREATE TABLE and CREATE INDEX statements which are normally generated. This option is set when performing the upgrade procedure described in section 6.5 *Upgrading the stored packages*. This option does not affect the behavior of UNIFACE when creating tables, indexes and packages on the fly.

The SQL script generated by the Create Table utility when this option is set can only be executed on a schema if the associated tables already exist in that schema. When you attempt to create the packages in a schema where the associated tables do not yet exist, obscure PL/SQL compilation errors occur.

disable checks

By default, when accessing a table, UNIFACE calls the PACKAGE_VERSION procedure in the stored package associated with the table once it has checked that the table exists. UNIFACE then checks whether the major package version of that package is compatible with the current major package version of the ORA driver, and stores the minor package version in its internal administration. This is explained in section 6.2 *Stored package version*.

If the driver option `disable checks` is set, UNIFACE does not call the PACKAGE_VERSION procedure. In that case, you must ensure that both the major package version and the minor package version are identical to the current major and minor package versions of the ORA driver. By disabling the check in this way, you improve performance.

Caution: If the `disable checks` *option is set, and either the major or minor package version is not identical to the current package version of the ORA driver, obscure errors can occur at run time. To avoid this problem, follow the package upgrade procedure described in section 6.5 Upgrading the stored packages, before setting the* `disable checks` *option. It is recommended that you first test your applications and databases without the* `disable checks` *option set whenever you install a new release of UNIFACE or the ORA driver.*

ignore missing packages

By default, the ORA driver generates an error when a stored package does not exist in the database. This happens when UNIFACE calls the PACKAGE_VERSION procedure after it has checked whether a table exists. If, however, the driver option `ignore missing packages` is set, and the call to the PACKAGE_VERSION procedure fails abnormally, UNIFACE ignores the package, does not raise an error, and continues as normal using only dynamic SQL for the table concerned.

You can selectively disable the use of stored packages by not creating them for some tables, and by setting the `ignore missing packages` option. This option is also useful when an application primarily accesses a database with the Procedural option, but it must also access a database without this option.

The disadvantages of the `ignore missing packages` option are:

- If a stored package is available, but an abnormal error occurs when calling the PACKAGE_VERSION procedure, the error goes unnoticed.
- When ignore missing packages is in use, you cannot use the disable checks option. This means that you do not gain the performance improvement provided by disable checks.

disable hint first_rows

By default, UNIFACE generates the optimizer hint first_rows for appropriate situations. The generated hint is also included in the SQL statements in the stored packages. Using hints in certain situations can have a negative impact on performance.

If the disable hint first_rows option is set, UNIFACE does not include the hint in the stored packages. This is true for the Create Table utility and when creating packages on the fly.

If you want to include the hint in some packages but not in others, you can do so by running the Create Table utility for individual tables and selectively setting the disable hint first_rows option.

6.6.1 Referential integrity

Implementation of relationships

When the foreign key rule of a relationship is RES or CAS, the relationship is implemented in ORACLE by a declarative integrity constraint:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name
FOREIGN KEY ( foreign_key_fields )
REFERENCES table_name ( referenced_key_fields )
[ON DELETE CASCADE]
```

The constraint name is the relationship name assigned in the Create Script utility.

When the foreign key rule of a relationship is NUL, the relationship is implemented by a set of triggers in table 6-1:

Table 6-1 Trigger names used to implement relationships.

Trigger name	Triggers on	Purpose
*one_table_name*_U	UPDATE on the one table.	Disallow update of referenced key.
*one_table_name*_D	DELETE on the one table.	Nullify referencing keys.
*many_table_name*_IU	INSERT or UPDATE on the many table.	Disallow unmatched foreign key.

The declarative foreign key constraint names and the names of database triggers follow the naming conventions for schema objects as described in section 3.3.2 *Naming rules for schema objects*.

When a table is involved in multiple relationships with a NUL foreign key rule, multiple actions are combined in the triggers. The name of the relationship is included in comment with every action.

The SQL generated by the Create Script utility to check the database for violations of referential integrity only checks the relationships with a NUL foreign key rule. The relationships with a RES or CAS foreign key rule are automatically checked by ORACLE when you attempt to create these relationships.

Note: The `ALTER TABLE ... ADD CONSTRAINT` *generated by UNIFACE does not include the* `EXCEPTIONS INTO` *clause. If you want ORACLE to report which rows violate referential integrity, you must create an exceptions table and edit the* `ADD CONSTRAINT` *statement generated by UNIFACE. Refer to your ORACLE documentation for more information.*

The SQL generated by the Create Script utility to check the database for referential integrity violations uses formatting features of the ORACLE utility SQL*Plus. Use SQL*Plus to execute the generated file.

Driver option to disable foreign key rules

By default, the UNIFACE CAS and NUL foreign key rules are enabled when using ORACLE. When using UNIFACE, it is recommended that you do not change this default in a development environment otherwise there is an overhead of the data conversion procedure.

If you decide to implement the relationships in ORACLE, you must set the following driver option in the USYS$ORA_PARAMS assignment:

```
USYS$ORA_PARAMS = disable foreign key rules
```

You can also use the short format:

```
USYS$ORA_PARAMS = df
```

Refer to chapter 10 *Overview of driver options (USYS$ORA_PARAMS)* for more information on the syntax of the USYS$ORA_PARAMS assignment.

Limitations

The following limitations apply when using the referential integrity enhancements with ORACLE:

- It is not feasible to create tables on the fly with referential integrity support.
- The first 26 characters of relationship names within an application model should be unique.
- UNIFACE does not provide SQL scripts for referential integrity on the data tables of the demo application. Do not set the `disable foreign key rules` driver option when accessing any of these tables in ORACLE.

There are some limitations on the use of triggers in ORACLE:

- The trigger for the nullify action is not valid when it attempts to nullify a foreign key in a mutating or constraining table. Refer to your ORACLE documentation for more information on mutating and constraining tables.
- Problems can arise when there is a cycle in the graph of relationships with at least one NUL foreign key rule. The simplest case is a self-referential table with a NUL foreign key rule. UNIFACE makes no attempt to detect or overcome these limitations and may create nonvalid triggers in such cases.

Relationships in ORACLE are only supported when the default packing code mapping is used. For this reason, the `disable foreign key rules` driver option cannot be set when any of the u2 packing code mapping options is set. When you do attempt to set both a u2 packing code mapping option and the `disable foreign key rules` option, the following error message is generated:

```
ORACLE Driver Error [-9]: Invalid combination of driver options in
USYS$ORA_PARAMS assignment
```

Attempting to use the Create Script utility with any of the u2 packing code mapping options set causes the following error:

```
ORACLE Driver Error [-90]: Relationships in ORACLE not supported with a u2 packing code mapping
```

If any of the relationships have the NUL foreign key rule, the ORACLE server must have the Procedural Database option because these are implemented by database triggers.

If any of the relationships have the NUL foreign key rule, the `CREATE TRIGGER` privilege is required to be able to create the relationships. The `CREATE TRIGGER` privilege should be given to the `UNIFACE_DEVELOPER` role.

Foreign key fields cannot have the Long, Long Raw, BLOB, or CLOB storage format.

When databases have been migrated from earlier ORA driver versions using the backward compatibility method rather than the conversion method, problems can occur when creating the relationships. This is because earlier ORA driver versions created a Unique Index with Not Null constraints for the primary key and candidate key, whereas relationships can only be defined on an ORACLE Primary Key or a Unique Key. The generic database conversion procedure should be used to export, re-create and import data, or tables should be modified manually to create the Primary Key and Unique Key constraints.

Chapter 7 Service Stored Procedures

This chapter discusses:

- Component names
- Parameters
- Exception behavior

Service Stored Procedure functionality is only available in the ORACLE 8 (U4.0 and U5.0) drivers.

For information about stored procedure components, see the *URB Interfaces Manual*.

7.1 Component names

The stored procedure name used for execution by UNIFACE is derived from the component name and the operation name as shown in table 7-1:

Table 7-1 Component name and component operation name specification.

Literal component name	Literal operation name	Stored procedure name
Not specified	Not specified	*4GLComponentName.4GLOperationName*
Not specified	Specified	*LiteralOperationName*†
Specified	Not specified	*LiteralComponentName.4GLOperationName*
Specified	Specified	*LiteralComponentName.LiteralOperationName*

Table notes:
† These stored procedures have no package. All other stored procedures are included within packages.

7.2 Parameters

Parameter names are not used during the execution of stored procedures. However, their ordering (that is, their relative position) is used for mapping parameters.

Parameter types

ORACLE supports multiple output entity parameters. You can use both basic and entity parameters for output. The maximum number of parameters is 100, of which 25 can be entity parameters.

Data conversions

The data types of operation parameters must closely match the stored procedure parameters. If necessary, explicit and implicit conversions are used to achieve an optimal mapping for all data types.

The supported parameter data type mappings for ORACLE are shown in table 7-2. (The data type mappings shown in table 7-2 apply to both basic parameters and entity parameters.) I

Table 7-2 UNIFACE to ORACLE parameter data type mappings.

ORACLE data type	String	Boolean	Numeric	Float	(L)Date	(L)Time	(L)Datetime	Raw	Image
char	Y	Y	Y	Y	†	†	†		
varchar2	Y	Y	Y	Y	†	†	†		
number	Y	Y	Y	Y	†	†	†		
float	Y		Y	Y					
date	Y				Y	Y	Y		
long									
lobs									

Table notes:

† This data type conversion depends on ORACLE behavior which can be changed by a query. For example:

```
alter session set NLS_DATE_FORMAT='DDMMYYYY HH24:MI:SS'
```

For information on `alter session` parameters, see your ORACLE documentation.

Input, output and in-output parameters

All parameter data is passed to ORACLE as binded data. To achieve an optimal mapping for all parameters, both ORACLE parameter data type and operator parameter data type information is used to determine the ORACLE binding data types.

NULL support

Empty input string values are mapped to ORACLE values as NULL values. There is one exception to this: empty UNIFACE boolean data types are always mapped to FALSE. This is standard functionality.

7.3 USYS$ORA_PARAMS

There are no specific driver settings for ORACLE SSPs. For information on USYS$ORA_PARMS options, see chapter 10 *Overview of driver options (USYS$ORA_PARAMS).*

7.4 Exception behavior

The exceptions shown in table 7-3 are detected by the ORACLE driver:

Table 7-3 Exception message numbers and meanings.

$procerror	Meaning
-21	Connection failure
-55	Input parameters failure
-56	Output parameter failure
-150	Driver or ORACLE failure
-166	Operation is not stateless
-1406	Memory allocation failure

The following messages are generated by the ORACLE driver when signature matching:

- Operation/Procedure name mismatch. This is detected by ORACLE, and is reported with the message:

  ORA-6550 Component <*stored procedure*> must be declared

- Signature mismatch in number of arguments. This is detected by ORACLE, and is reported with the message:

  ORA-1123 Wrong number of parameters

- Entity parameters with non-matching number of fields. This is detected by ORACLE, and is reported with the message:

  ORA-1007 A reference was made to a variable not listed in the SELECT clause.

7.5 Examples

The following examples illustrate the use of Service Stored Procedures. The first example shows a Service Stored Procedure with two basic parameters. It is executed with the following `activate` Proc statement:

```
activate "EXAMPLE".sp_insert( $in$, $inout$ )
```

The sp_insert operation has the parameters shown in table 7-4:

Table 7-4 sp_insert operation parameters.

Name	Type	Data Type	Input/Output
u_str	Basic	String	Input
u_num	Basic	Numeric	Input/Output

The ORACLE SQL is as follows:

```
CREATE OR REPLACE
PACKAGE BODY "WORKTODO".EXAMPLE
as

procedure sp_ent (
    entcur1 IN OUT ent1)
is
BEGIN
open entcur1 for select str, num from tab;
END;
```

```
procedure sp_insert (
    str IN     varchar2,
    num IN OUT number)
is
BEGIN
insert into tab values (str, num);
num := num + 1;
END;
END;
/
```

The second example shows the use of a Service Stored Procedure with one entity parameter. It is executed with the following `activate` Proc statement:

```
activate "EXAMPLE".sp_ent( "tab.model")
```

The sp_ent operation has the parameters shown in table 7-5:

Table 7-5 sp_ent operation parameters.

Name	Type	Data Type	Input/Output
u_ent	Entity	-	Output

The tab entity has the fields shown in table 7-6:

Table 7-6 tab entity structure.

Name	Type	Data Type
str	S	VC40
num	N	C20

The ORACLE SQL script is as follows:

```
CREATE TABLE tab (str varchar2(40), num number(20));
/
CREATE OR REPLACE
PACKAGE "WORKTODO".EXAMPLE
as
cursor c1 is select str, num from tab;
type ent1 is ref cursor return c1%ROWTYPE;

procedure sp_ent (
    entcur1 IN OUT ent1);

procedure sp_insert (
    str IN     varchar2,
    num IN OUT number);

END;
/
```

Chapter 8 Data types and packing codes

Table 8-1 describes the default mapping of UNIFACE packing codes to ORACLE storage formats:

Table 8-1 How UNIFACE packing codes map to ORACLE storage formats. ***part 1 of 2***

UNIFACE packing codes	ORACLE storage format
B1-B4	Char
C1-C255	Char
C256-C2000	Varchar2
C2001-C*n*	Long
C*	Long (Long Raw if binary length ID is used)
D	Date
D1-D13	Date
E	Date
E1-E13	Date
F4, F8	Number
I1-I8	Number
M1-M4	Number
N1-N32	Number
(N)C1-32[†]	Number
O1-O32	Number
P1-P16	Number
Q1-Q16	Number
R1-R255	Raw
R256-R*n*	Long Raw

Table 8-1 How UNIFACE packing codes map to ORACLE storage formats. ***part 2 of 2***

UNIFACE packing codes	ORACLE storage format
R*	Long Raw
SC1-SCn	Long in U3.*x*, CLOB in U4.0 and U5.0
SC*	Long in U3.*x*, CLOB in U4.0 and U5.0
SR1-SRn	Long Raw in U3.*x*, BLOB in U4.0 and U5.0
SR*	Long Raw in U3.*x*, BLOB in U4.0 and U5.0
SU1-SUn	Long Raw in U3.*x*, CLOB in U4.0 and U5.0
SU*	Long Raw in U3.*x*, CLOB in U4.0 and U5.0
T	Date
T1-T3	Date
U1-U255	Char
U256-U2000	Varchar2
U2001-U*n*	Long
U*	Long (Long Raw if binary length ID is used)
VC1-VC2000	Varchar2
VC2001-VC*n*	Long
VC*	Long (Long Raw if binary length ID is used)
VR1-VR255	Raw
VR256-VR*n*	Long Raw
VR*	Long Raw
VU1-VU2000	Varchar2
VU2001-VU*n*	Long
VU*	Long (Long Raw if binary length ID is used)
Y1.0-Y32.9	Raw (Long Raw if length > 255)
Z	Raw (Long Raw if length > 255)

Table notes:
† Numeric data type packed with C packing code.

8.1 Explanation of ORACLE storage formats

The information that follows is only intended as a guide and is not a complete description of ORACLE storage formats. If you are in any doubt about the exact requirements in your environment, see the ORACLE documentation.

8.1.1 Char

The Char storage format is used to store fixed-length character strings. You may specify up to a maximum of 255 bytes for a Char column (the default length is one byte). When storing Char columns, ORACLE pads them out with blanks to their maximum length. When it compares two Char column values, ORACLE uses blank-padded comparison semantics.

The character set used by the database is defined at the time the database is created. See section 4.8 *ORACLE character sets*, for information on configuring the ORACLE character sets.

8.1.2 Varchar2

The Varchar2 storage format is used to store variable-length character strings. You may specify up to a maximum of 2000 bytes for a Varchar2 column (the default length is one byte). ORACLE uses variable-length storage techniques when it stores values in Varchar2 columns. When it compares two Varchar2 column values, ORACLE uses non-padded comparison semantics.

8.1.3 Varchar

The Varchar storage format is synonymous with the Varchar2 storage format. UNIFACE does not create columns in ORACLE with the Varchar storage format. You can however access existing Varchar columns by defining them in UNIFACE as you would define Varchar2 columns. For example, in UNIFACE, you should define a Varchar(40) column as having data type S and packing code VC40.

8.1.4 Number

Number is a storage format for fixed and floating point numbers. The values you can store range from -9.99...* 10^{125} through 9.99...* 10^{125}, and can have a maximum length of 38 significant digits.

UNIFACE allows you to specify non-default decimal characters by setting the ORACLE parameter `NLS_NUMERIC_CHARACTERS` (for example, if you want to use a character other than the period (`.`) to represent the decimal point). This does not change the way UNIFACE interprets and presents numeric values, except in the `sql` Proc instruction, the SQL Workbench and the `read where` Proc instruction.

8.1.5 Date

The Date storage format is used to store point-in-time values, which consist of both a date and a time. The values which can be stored in a Date field range from Jan 1, 4712 BC 00.00.00 through Dec 31, 4712 AD 23.59.59.

If no time is available with a value to be stored in the Date storage format, UNIFACE adds the default time of 00:00:00. If time is recorded without date information, UNIFACE adds the NULL date of 01011900 (1 January, 1900). Arithmetic functions with Date are possible.

UNIFACE allows you to specify non-default date formats by setting ORACLE parameters such as `NLS_DATE_FORMAT`. This does not change the way UNIFACE interprets and presents Date values, except in the `sql` Proc Instruction, the SQL Workbench and the `read where` Proc instruction.

8.1.6 Long

The Long storage format is used to store variable-length character data to a maximum length of two gigabytes. The following restrictions apply when using the Long storage format:

- An ORACLE table may contain only one Long or Long Raw column.
- Stored procedure arguments cannot be declared using the Long storage format.
- Long and Long Raw columns cannot appear in integrity constraints (except for NOT NULL).

- Long columns cannot be indexed.
- Long columns cannot appear in the `where` clause, the `order by` clause, nor in SQL functions, expressions and conditions.
- Segmented read is possible with Long columns, but segmented write is not.

UNIFACE uses the Long storage format in the following circumstances:

- When a field is defined in UNIFACE as having data type S or SS and packing code C,VC, U, or VU, and that field exceeds the maximum length of the Varchar2 storage format, UNIFACE uses the Long storage format. UNIFACE cannot store or retrieve values which exceed the length defined for the field in the application model. This restriction is not enforced by ORACLE, however, as Long columns do not have a declared maximum length.
- For BLOB data when using a U3.*x* driver. UNIFACE fields defined with data type S or SS, and packing code SC or SU, are stored using the Long storage format and can be used to store BLOB data. When you define fields with the segmented packing codes SC or SU, UNIFACE can use them to store data to a maximum length of 2 gigabytes. This is, however, a theoretical limit, and the true limit depends upon the constraints imposed by your operating system. See section 4.5 *Implementation of segmented interfaces*, for more information on using segmented packing codes with ORACLE.
- For variable-length fields. If you define a variable-length field in UNIFACE, it is stored in a Long or Long Raw column of an overflow table (this is according to UNIFACE's own variable-length techniques).

When using the U3.*x* driver, if you want to access a Long column in an existing database, you must use the S data type and the SC* packing code. When using the U4.0 or U5.0 driver, you must use the S data type and the VC* packing code.

8.1.7 Raw

The Raw storage format is used to store binary data which ORACLE does not convert or interpret. You may specify a length up to a maximum of 255 bytes for a Raw column.

8.1.8 Long Raw

The Long Raw storage format is used to store large amounts of binary data which ORACLE does not convert or interpret. You can store up to two gigabytes (the theoretical limit) of data in a Long Raw column. The ways in which UNIFACE uses the Long Raw storage format, and the limitations which apply to its use are the same as those described for the Long storage format.

When using the U3.*x* driver, if you want to access a Long Raw column in an existing database, you must use the R data type and the SR* packing code. When using the U4.0 or U5.0 driver, you must use the R data type and the VR* packing code.

8.1.9 CLOB and BLOB

These are used to store CLOB and BLOB data, respectively, where UNIFACE fields are defined with data type S or SS and packing code SC. When you define fields with the segmented packing code SC, UNIFACE can store data to a maximum length of four gigabytes. These fields are treated like any other except that they may not be used in a condition, either explicitly, through a `u_where` or `where` Proc, or implicitly, by entering data in a CLOB or BLOB field before requesting a retrieve.

8.1.10 Rowid

Rowids are the physical addresses of rows in the database. UNIFACE uses Rowids because they are the fastest way to access individual rows. This speed of access becomes particularly important when UNIFACE must address a row several times in short succession (for example, the first time to select a row with its Rowid, the second time to lock that row and the third time to update that row).

UNIFACE uses Rowids automatically. It is neither necessary or possible to define the Rowid pseudo column of a table in the application model.

Entities that are accessed through ORACLE Gateway products use Rowids that vary from zero through 255 bytes. UNIFACE cannot currently handle variable-length Rowids but the `disable rowid` driver option can be set so that the driver can handle these entities.

8.1.11 Mlslabel and Raw Mlslabel

These two storage formats are only available with Trusted ORACLE and are therefore not supported by UNIFACE.

8.2 Modify packing code mapping

The UNIFACE U2.*x* drivers (where *x* is 1 or 0) supported several different ways to map packing codes to ORACLE storage formats. With the U3.*x* driver for ORACLE, it is recommended that you only use the default packing code mapping, and you should migrate existing databases to this method. See chapter 12 *Migrating between U2.x, U3.x, U4.0, and U5.0*, for more information on the migration procedure.

Note: If possible, use the default packing code mapping with the U3.x driver. The mapping code methods: `u2 default mapping`*,* `u2 enhanced_mapping`*,* `u2 enhanced_mapping_2` *and* `map fixed-length to variable` *are only provided for reasons of backward compatibility. They may be discontinued in a future version of UNIFACE, so you should migrate your database to use the U3.x default mapping at the earliest opportunity. There are no plans to discontinue the* `disable segmented field` *option.*

You can use an existing database which uses U2.*x* packing code mapping, with the U3.*x* driver. For this reason, the following driver options are available with the U3.*x* driver:

- `u2 default mapping`–When this option is set, the U3.*x* driver maps packing codes to storage formats using the U2.*x* driver's default mapping.
- `u2 enhanced_mapping`–When this option is set, the U3.*x* driver maps packing codes to storage formats using the U2.*x* driver's enhanced mapping. See the *DBMS Specific* module *ORACLE Driver Guide* for more information on U2.*x* enhanced_mapping.
- `u2 enhanced_mapping_2`–When this option is set, the U3.*x* driver maps packing codes to storage formats using the U2.*x* driver's `enhanced_mapping_2`. See the *DBMS Specific* module *ORACLE Driver Guide* for more information on U2.*x* `enhanced_mapping_2`.

Caution: If you are using one of these alternative mapping methods, or if you use the `disable segmented fields` *option (see below), you must convert all existing data if you decide to change to a different mapping method. See chapter 11 Generic database conversion procedure, for more information on the data conversion procedure.*

8.2.1 The map fixed length to variable option

When the `map fixed length to variable` option is set, all fields which would normally be mapped to the Char storage format (using U3.*x* default mapping), are instead mapped to the Varchar2 storage format. Use this option to avoid incompatibility when migrating from the U2.*x* driver to the U3.*x* driver. You cannot use this option if you have already specified one of the U2.*x* mapping options described above. See section 12.1.4 *Compatibility problems* for more information.

8.2.2 The disable segmented fields option

This option is useful only when using a U3.*x* driver. The `disable segmented fields` is another UNIFACE driver option which changes the way certain packing codes are mapped. When this option is set, UNIFACE transparently maps fields with segmented packing codes into the variable portion of the record, using UNIFACE's own variable-length techniques.

You can use this option in the following situations:

- If a table contains more than one Long or Long Raw column because you have defined one or more segmented fields in your application model.

  ORACLE does not allow more than one Long or Long Raw column per table. If you specify use of the UNIFACE variable-length technique (using overflow tables), you avoid this problem because only one Long or Long Raw column is necessary per table.
- If memory limitations specific to your operating system stop your application from writing a large segmented field. (See section 4.5 *Implementation of segmented interfaces* for more information on segmented I/O in ORACLE.)

If you specify using the UNIFACE variable-length technique, the driver is no longer dependent on operating-system dynamic memory allocation when writing a field with a segmented interface.

You cannot use this option if you have already specified one of the U2.*x* mapping options described above.

Caution: When you store fields in the database using the UNIFACE variable-length technique, they cannot be accessed by tools other than UNIFACE.

Chapter 9 System parameters

A number of environment variables are read from the environment and interpreted by ORACLE. The details of these variables, and how to define them for your operating system, can be found in your ORACLE documentation. For example, ORACLE on Alpha OpenVMS AXP uses a command file ORAUSER_*dbname*.COM, and ORACLE on UNIX uses shell-specific scripts, such as **oraenv**.

The information in this chapter is only intended as a guide. Refer to your ORACLE documentation for your operating system for more information.

9.1 Environment variables

The environment variables described in this section are the variables which are most important when using UNIFACE with ORACLE.

If you are using UNIFACE with the ORACLE network product SQL*Net, define the environment variables in the client environment. When you use PolyServer, they must be defined in the PolyServer environment.

9.1.1 ORACLE_HOME

The ORACLE_HOME (UNIX, MS-DOS, OS/2) environment variable must identify the location of the ORACLE installation in the file system. Whenever you install a specific ORA driver version, or when you link UNIFACE Seven, PolyServer or UNIFACE applications with a specific ORA driver version on UNIX, ORACLE_HOME must identify an installation of the correct ORACLE version. See chapter 1 *Introduction* for more information on the different ORA driver versions.

9.1.2 ORACLE_SID

The ORACLE_SID (UNIX) environment variable, in combination with ORACLE_HOME, identifies an ORACLE database. This is the default database to which the application connects when no database is specified in the logon path specification.

9.1.3 TWO_TASK

The TWO_TASK (UNIX) environment variable may specify a default ORACLE two-task communication driver (including SQL*Net drivers), a host and a database. This is the default database to which the application connects when no database is specified in the logon path specification. The TWO_TASK variable overrides the ORACLE_HOME and ORACLE_SID variables in determining the database to which the application will connect.

9.1.4 NLS_LANG

The NLS_LANG (UNIX) environment variable optionally specifies the language, territory, and character set National Language Support parameters. It is recommended that you define NLS_LANG when the database to which the application connects uses a different character set than the default character set assumed in your ORACLE client environment.

See section 4.8 *ORACLE character sets* for more information on specifying a character set.

Chapter 10 Overview of driver options (USYS$ORA_PARAMS)

This section presents a complete overview of all driver options, and describes the syntax of the USYS$ORA_PARAMS assignment.

Every option can be specified using either a Long format, which is readable, or a short format, which allows you to set many options in the USYS$ORA_PARAMS assignment. The assignment cannot be longer than one line in the assignment file.

*Note: If PolyServer is used, the USYS$ORA_PARAMS assignment setting must be in an assignment file on the server platform. The setting must be in an assignment file on the client platform if ORACLE is accessed locally or if SQL*Net is used.*

Most options are enable or disable options; that is, a specific feature is enabled or disabled by setting the option. Some options have a numeric parameter. This parameter must be a nonnegative decimal integer. The range of valid values is option-specific. Every numeric option has a default value.

The syntax rules for USYS$ORA_PARAMS are as follows:

- Each option may be specified once, either in Long format or in short format, not both. Options in Long format may be mixed with other options in short format.
- The order in which the options are specified is not significant.
- Options must be separated by a comma.
- Words and numbers within one option are separated by one or more spaces and tabs.
- Options are not case-dependent.

The valid options are listed in table 10-1:

Table 10-1 Driver options for the ORA driver. ***part 1 of 3***

Long format	Short format	Description
u2 default mapping	u2dm	Use packing code mapping compatible with default mapping of U2.1 ORA/ORT driver. See section 8.2 for more information. Only supported to facilitate migration. May be discontinued in future versions of UNIFACE.
u2 enhanced_mapping	u2em	Use packing code mapping compatible with enhanced_mapping of U2.1 ORA/ORT driver. See section 8.2 for more information. Only supported to facilitate migration. May be discontinued in future versions of UNIFACE.
u2 enhanced_mapping_2	u2em2	Use packing code mapping compatible with enhanced_mapping_2 of U2.1 ORA/ORT driver. See section 8.2 for more information. Only supported to facilitate migration. May be discontinued in future versions of UNIFACE.
map fixed length to variable	fv	Map all character strings to VARCHAR2. Do not use CHAR. Only supported to facilitate migration. May be discontinued in future versions of UNIFACE. See section 8.2 for more information.
disable packages	dpa	Do not generate or use stored packages. See section 6.6 for more information.
ignore missing packages	im	Use dynamic SQL when package is missing. See section 6.6 for more information.
disable segmented fields	dsf	Disable BLOB support. Map fields with segmented interface into the variable portion of the record. See section 8.2 for more information. This option is ignored by the U4.0 and U5.0 drivers because the drivers handling of segmented fields makes it redundant.
upgrade packages	up	Generate only packages in CTU, not tables and indexes. See section 6.6 for more information.

Table 10-1 Driver options for the ORA driver. ***part 2 of 3***

Long format	Short format	Description
array fetch size <number>	af <n>	Minimum size of array when using ORACLE array fetching. Default is platform-specific (typically 10). Range of valid values is 1 through 32767. Causes dynamic memory allocation of approximately (<n> + 1) * 15 kilobytes. Refer to section 4.6.2 for more information.
fixed array size	fa	Use exactly array size specified with option 'array fetch size'. Do not increase array size for small records. See section 4.6.2 for more information.
step size <number>	ss <n>	Recommended step size for UNIFACE stepped hitlist mechanism. The range of valid values is from 0 through 32767. 0 means that the stepped hitlist mechanism is disabled. See section 4.6.2 for more information.
disable hint first_rows	dhfr	Do not generate the optimizer hint FIRST_ROWS.
disable escape	de	Do not use ESCAPE clause with LIKE operator. Causes incorrect behavior of retrieve profiles. See section 4.3.1 for more information. Only supported to facilitate migration. May be discontinued in future versions of UNIFACE.
support obsolete 3gl services	os	Respond to ulda and uopenflag set by user-defined 3GL. Only supported to facilitate migration. May be discontinued in future versions of UNIFACE.
disable precompiler connect	dpc	Use OCI instead of precompiler interface for first logon path. This option is ignored by the U4.0 and U5.0 drivers because they cannot connect using the precompiler interface.
disable checks	dc	Skip the run-time consistency checks for the storage formats when the table is opened, and for the major or minor version when packages are used. Use with care. See section 3.3.5 and section 6.6 for more information.

Table 10-1 Driver options for the ORA driver. ***part 3 of 3***

Long format	Short format	Description
open cursors <number>	oc *n*	Maximum number of cursors per logon path which may be explicitly opened. Default is 45. Absolute minimum is four. Realistic minimum is two per open table plus one per segmented field. Maximum is the value of the OPEN_CURSORS initialization parameter of the ORACLE server, minus some cursors to account for recursive cursors opened by ORACLE. See section 4.6.1 for guidelines.
disable foreign key rules	df	Disabling the UNIFACE implementation of the CAS and NULL foreign key rules on delete of a row in a one table. ORACLE handles the referential integrity rules.
disable rowid	dri	Disable the use of Rowids. The driver uses the primary key to identify a record. This enables the driver to handle entities which are accessed through ORACLE gateways.
multi byte	multi byte	Enables ORACLE for multibyte character sets.

The following are the limitations on combining driver options:

- Only one of u2 default mapping or u2 enhanced_mapping or u2 enhanced_mapping_2 can be set at any one time.
- When u2 default mapping or u2 enhanced_mapping or u2 enhanced_mapping_2 is set, the following driver options are either already implied or not applicable. They cannot, therefore, be specified. They are:

  disable packages
  disable segmented fields
  ignore missing packages
  upgrade packages
  map fixed length to variable
- The following combinations of options are not valid:

  disable packages and ignore missing packages
  disable packages and upgrade packages
  disable checks and ignore missing packages

The following driver errors may be generated because of incorrect assignments to USYS$ORA_PARAMS:

```
ORACLE Driver Error [-7]: Value out of range: value

ORACLE Driver Error [-8]: Syntax error or illegal value in USYS$ORA_PARAMS
assignment

ORACLE Driver Error [-9]: Invalid combination of driver options in
USYS$ORA_PARAMS assignment
```

Some examples of correct USYS$ORA_PARAMS assignments follow:

```
USYS$ORA_PARAMS = disable packages, oc 100, af 12

USYS$ORA_PARAMS = step size 20, array fetch size 20, fixed array size

USYS$ORA_PARAMS = dhfr, fa, oc 100, de, im, dsf
```

10.1 multi byte

The `multi byte` driver option enables entity and field names to be stored in a multibyte language (such as Japanese). This driver option is not required to store double-byte data within single-byte entity and field names. The multi byte driver option has the following format:

`USYS$ORA_PARAMS multi byte`

See the ORACLE documentation for more information.

When the `multi byte` option is set, DBMS wildcard literals are not supported. This option has no effect on UNIFACE wildcards.

The difference between a DBMS wildcard literal and a UNIFACE wildcard is that a UNIFACE wildcard is translated into the DBMS wildcard. Wildcard literal support means that if a DBMS wildcard is included in the actual data, it is not treated as a wildcard.

In other words, if the actual data includes a percent sign (%), and wildcard literals are supported, the % in the data is treated like a percent sign and not a wildcard by the DBMS.

Chapter 11 Generic database conversion procedure

This section describes a generic database conversion procedure. This procedure should be used in the following circumstances:

- When migrating from one ORA driver version to another (usually from U2.*x* to U3.*x*). You should only perform database conversion if the migration procedure in chapter 12 *Migrating between U2.x, U3.x, U4.0, and U5.0* instructs you to do so.
- When setting or removing one of the following ORA driver options:

```
u2 default mapping
u2 enhanced_mapping
u2 enhanced_mapping_2
map fixed length to variable
disable segmented fields
```

- You only need to convert the tables which are affected when you change the driver option. As this may include tables in the Repository and UNIFACE Runtime tables (for example, UOBJ), it can be more convenient for you to perform the complete conversion on all tables.
- If you need to overcome incompatibilities at the database I/O level (which can occur when upgrading the UNIFACE version). Only perform database conversion when you have been instructed to do so.

11.1 The conversion procedure

Note: This conversion procedure converts in two stages: from the existing database to the UNIFACE TRX transfer format, and from TRX to the desired database. See the UNIFACE online help for more information on the utilities available for deployment.

The conversion procedure distinguishes between the old environment— the one which you have been using to access the existing databases— and the new environment, the one which you want to start using. The environment means to the following:

- The specific version of UNIFACE
- The specific ORA driver version
- The packing code mapping used and the setting of the driver options which modify the packing code mapping
- The specific version of ORACLE

The main stages of the conversion procedure are summarized as follows:

1. Export all data to the UNIFACE TRX transfer format.
2. Switch to the new environment.
3. Import all data from the UNIFACE TRX format.

Note: When you switch from the old to the new environment, you may have to change environment variables (for example, `$usys`, `ORACLE_HOME`, `ORACLE_SID`), install a new version of ORACLE, create a new database, install a new version of UNIFACE or install a new version of the ORA driver. It is strongly recommended that you install, set up, and test the new environment before starting the conversion procedure. Make sure that the old and the new environments can coexist, and that you can switch smoothly between them.

11.1.1 The full conversion procedure

To carry out the conversion procedure:

1. Identify the tables to be converted. These may include tables in the Application Objects Repository. See the *UNIFACE Reference Manual* for more information.
2. Back up all tables which need to be converted. You may consider this redundant, as the conversion procedure does not destroy old data, and the export to TRX format is a backup mechanism in itself. It is recommended to use this backup only as an extra safety precaution.
3. Switch to the old environment.
4. If the Repository is stored in ORACLE, use the Export Repository Objects form by selecting Tools–>Export to export the Repository to TRX.

5. Perform `analyze model` on all application models describing the data you want to convert. You must do this before exporting the data.
6. Export all tables which need to be converted, except for UOBJ and the tables from the Repository. To do this, use the Deployment–>Database Utilities–>Convert Data form, or the `/cpy` command line switch. For example, to export the `DEPT.PERSONNEL` table, you must enter the following on the command line:

   ```
   /cpy ora:dept.personnel trx:
   ```

7. Switch to the new environment.
8. Create the Repository in ORACLE, using the SQL scripts provided in the distribution kit. See section 2.1 *SQL scripts for creating the Application Objects Repository* for more information.
9. If the Repository was exported, as in step 4, use the Tools–>Import form to import the Repository from TRX. Run `/con` and `/all` on the command line to analyze the application models and to compile all global objects. See the *UNIFACE Reference Manual* for more information on these command line switches.
10. Perform analyze model on all application models describing the tables you want to re-create. You must do this before using the Create Table utility.
11. Use the Create Table utility to generate SQL scripts which will create your databases. Note that a number of driver options affect the behavior of the Create Table utility with the ORA driver. See section 2.2.1 *Create Table utility*, for more information. Use one of the ORACLE utilities SQL*Plus or SQL*DBA to execute the generated script.
12. Import the tables that you exported in step 6 using the Deployment–>Database Utilities–>Convert Data form, or the `/cpy` command line switch. For example, to import the `dept.personnel` table, you must enter the following on the command line:

    ```
    /cpy trx:dept.trx ora:
    ```

13. Depending on how UOBJ was used, you may need to recompile global definitions using `/all` on the command line. See the *UNIFACE Reference Manual* for more information.
14. Rebuild and test your applications in the new environment. Note that installing a new ORA driver version may require that you relink all applications which use the driver. However, this depends on your operating system.

Chapter 12 Migrating between U2.*x*, U3.*x*, U4.0, and U5.0

This section explains the migration procedure from the U2.*x* ORA/ORT driver to the U3.*x* drivers, as well as the migration from U3.*x* to the U4.0 and U5.0 drivers. The overall structure of the migration procedure is described below. Many references are given to other sections such as for detailed information on conversion steps, driver options for backward compatibility and so on.

Use this chapter as your guide through the migration procedure.

12.1 Migrating from U2.*x* to U3.*x*

This section describes the migration procedure to migrate from a U2.*x* driver to a U3.*x* driver.

12.1.1 The migration procedure

The description of the migration procedure given here assumes that both the U2.*x* and the U3.*x* ORA driver are installed and available in your environment. Verify that they are installed before proceeding with the migration procedure.

Preparing to migrate

Before carrying out the migration procedure, you must complete the following steps:

1. Review your application models and applications for incompatibilities which can arise. In most cases, you can solve incompatibilities either by upgrading your application model or application, or by setting a specific driver option for backwards-compatible behavior. See section 12.1.4 *Compatibility problems*, for a detailed description.
2. Review and modify assignment files, as the U2.*x* and U3.*x* drivers are incompatible with respect to assignment files. See section 12.1.5 *Review and modify assignment files* for more information.

Depending on which driver you are currently using, read the relevant section on the methods you can use to overcome the incompatibility between your existing databases and the U3.*x* ORA driver.

12.1.2 Migrating from the U2.0 ORA/ORT driver

If you are currently using the U2.0 ORA/ORT driver with ORACLE6, choose one of the following methods to overcome the incompatibility between your existing databases and the U3.*x* driver:

- The conversion method
- The backward-compatibility method

The conversion method

Convert your existing databases using the generic database conversion procedure. In this case, all new functionality will be available.

Using this method, perform the following two migrations at the same time:

- Migration from ORACLE6 to ORACLE7.*x* or 8
- Migration from the U2.0 ORA/ORT driver to the U3.*x* driver

See chapter 11 *Generic database conversion procedure* for a full description of converting databases.

Backward-compatibility method

Perform the following steps to use the backward-compatibility method:

1. Migrate from the U2.0 ORA/ORT driver with ORACLE6 to the U2.1 ORA/ORT driver with ORACLE7.*x* or 8.

2. Set one of the following driver options with the U3.*x* driver, which makes it fully backward compatible with the U2.1 driver with respect to database I/O:

   ```
   u2 default mapping
   u2 enhanced_mapping
   u2 enhanced_mapping_2
   ```

   Choose the option which is consistent with the option in use by the U2.1 driver. See chapter 10 *Overview of driver options (USYS$ORA_PARAMS)* for information on the consistent option combinations.

If you migrate in this way, both stored package and BLOB functionality are not supported in your new database. When you use this migration path, the U2.1 ORA/ORT and the U3.*x* ORA drivers can be used interchangeably to access the same database.

Note: It is strongly recommended that you use the conversion method. The only advantage to the backward-compatibility method is that you may be able to avoid a generic database conversion procedure in some circumstances. The migration procedure for the backward-compatibility method is much more complex.

12.1.3 Migrating from the U2.1 ORA/ORT driver

If you are currently using the U2.1 ORA/ORT driver with ORACLE7.*x* or 8, choose one of the following methods to overcome the incompatibility between your existing databases and the U3.*x* driver:

- The conversion method
- The backward-compatibility method

The conversion method

Convert your existing databases using the generic database conversion procedure. In this case all new functionality will be available. Once you have performed this conversion, the U2.1 driver can no longer be used to access your databases.

See chapter 11 *Generic database conversion procedure* for a full description of converting databases.

The backward-compatibility method

Set one of the following driver options with the U3.*x* driver, which makes it fully backward compatible with the U2.1 driver with respect to database I/O:

```
u2 default mapping
u2 enhanced_mapping
u2 enhanced_mapping_2
```

Choose the option which is consistent with the option in use by the U2.1 driver. See chapter 10 *Overview of driver options (USYS$ORA_PARAMS)*, for information on the consistent option combinations.

If you migrate in this way, both stored package and BLOB functionality are not supported in your new database. When you use this migration path, the U2.1 ORA/ORT and the U3.*x* driver can be used interchangeably to access the same database.

Note: It is strongly recommended that you use the conversion method. However, you may want to use the backward-compatibility method to test your applications with the U3.x ORA driver and investigate compatibility problems.

12.1.4 Compatibility problems

There are compatibility problems which can arise when migrating from the U2.*x* to the U3.*x* driver.

$dberror

You must review all applications which rely on the interpretation of ORA driver errors returned in the `$dberror` Proc variable. If your applications do not rely on `$dberror`, you can ignore this.

Driver error numbers returned in `$dberror` are defined independently for the U2.*x* and U3.*x* driver. Applications which rely on the interpretation of driver errors in `$dberror` must be reviewed and modified for use with the U3.*x* driver. See chapter 14 *Error messages* for an overview of the most important driver error messages.

It is recommended that you do not design applications which rely on the interpretation of specific driver error numbers in `$dberror`, as such errors are not generally meaningful to an application.

ORACLE error numbers are passed unchanged in `$dberror`, both with the U2.*x* and the U3.*x* driver.

Fixed-length character strings

This section refers to incompatibilities caused by the conversion method of migration. If you use the backward-compatibility method, you can ignore this section.

By default, the ORA U3.*x* driver maps the C and U packing codes with the S and SS data types to the fixed-length Char storage format, and it maps the VC and VU packing codes to the variable-length Varchar2 storage formats. The U2.*x* driver, however, maps packing codes to the variable-length Char storage format in ORACLE6 and to the variable-length Varchar2 storage format in ORACLE7.3 and 8. This can unintentionally result in variable-length Char columns in ORACLE6.0, or variable-length Varchar2 columns in ORACLE7.*x* or 8, being converted to fixed-length Char columns in ORACLE7.*x* or 8 when a database is converted from the U2.*x* driver to the default mapping of the U3.*x* driver.

The best solution to this problem is to change the packing code definitions in the application model from C to VC, and U to VU, since this reflects the intended mapping to variable-length storage techniques.

Do this by changing the packing code in the application model *before* you perform the generic database conversion (see chapter 11 *Generic database conversion procedure* for more information).[1] This procedure is quite safe as the C and VC (and U and VU) packing codes are equivalent when used with the U2.*x* driver.

Alternatively, the `map fixed length to variable` driver option can be set with the U3.*x* ORA driver, causing all fields to be mapped to the Varchar2 storage format that would otherwise be mapped to the Char storage format.

The `map fixed length to variable` driver option is supported solely for the purpose of allowing a simple migration to the U3.*x* ORA driver, and it may be discontinued in a future version of UNIFACE. It is recommended that you change the application model packing code definitions of string fields (S and SS data types) from C and U to VC and VU, respectively, when the variable-length storage technique is intended.

1. If you wait until after you have created the database, you must change the packing code in the application model *and* carry out data conversion from the Char to the Varchar2 storage format in the ORACLE database.

Segmented fields

This section refers to possible incompatibilities caused by the conversion method of migration when your application models include segmented fields. If you use the backward-compatibility method, you can ignore this section.

By default, the U3.*x* ORA driver maps segmented field packing codes to the Long or Long Raw storage formats (BLOB support). This can cause incompatibility problems, as the Long and Long Raw storage formats have the following limitations:

- There can be only one Long or Long Raw field per base table. For example, an entity in an application model with both an S,SC* and an S,C* field caused no problems with the U2.*x* ORA/ORT driver, as UNIFACE mapped both fields into one Long field using UNIFACE proprietary variable-length techniques. With the U3.*x* driver, however, the table would have two Long columns, causing the driver to generate the following error when an attempt is made to create the table:

  ```
  ORACLE driver error [-87]: The table would contain more than one LONG or
  LONG RAW column.
  ```

- ORACLE supports segmented read access to Long and Long Raw fields, but does not support segmented write access. For that reason, UNIFACE must use operating-system dynamic memory allocation to obtain a storage area as large as the complete segmented field (theoretical limit two gigabytes), when a segmented field is being inserted or updated. Since there may be platform-specific memory limitations, this may cause applications which worked perfectly with the U2.*x* driver to fail.

These compatibility problems can be overcome by setting the driver option `disable segmented fields` with the U3.*x* ORA driver, which causes segmented fields to be mapped into the UNIFACE proprietary variable-length techniques, in the same way that the U2.*x* driver does.

Array fetching

When working with the U3.*x* driver, UNIFACE allocates additional memory to improve performance. This section contains information on reducing memory usage, rather than improving performance.

By default, UNIFACE uses ORACLE array fetching with a value of at least 10 for the array size (on most operating systems) when the stepped hitlist is being built. This reduces the client/server communication overhead, at the cost of additional memory usage by the ORA driver. With the U2.*x* driver, array fetching is used only to a very limited extent, without making use of additional memory.

If you want to reduce memory usage rather than communication overhead, the `array fetch size` *number* driver option can be set to reduce the array size to the minimum value of 1. This still allocates some additional memory.

Stepped hitlist mechanism

You can avoid a change in the behavior of the UNIFACE stepped hitlist mechanism when migrating from the U2.*x* ORA/ORT driver to the U3.*x* driver.

By default, the U3.*x* ORA driver makes optimal use of the memory that is allocated for ORACLE array fetching. The size of the allocated memory is calculated based on the largest possible record. When smaller records are fetched, the array fetch size is automatically increased to reduce the client/server communication overhead as much as possible. Since the step size of the UNIFACE stepped hitlist must be a multiple of the actual array fetch size computed, the step size can be very large. Furthermore, since the actual array size depends on the size of the record, the step size may vary for different tables.

This side effect of large and varying step sizes does not occur with the U2.*x* driver.

The `fixed array size` option can be set to force UNIFACE to use exactly the array size specified with the `array fetch size` option, rather than computing the maximum array size dynamically.

ESCAPE clause

There can be incompatibility when existing applications rely on the use of ORACLE wildcard characters in UNIFACE retrieve profiles.

By default, the U3.*x* driver uses the ESCAPE clause with the LIKE operator to stop ORACLE wildcard characters, which occur as literal characters in UNIFACE retrieve profiles, from being interpreted. The U2.*x* driver, however, does not generate the ESCAPE clause. Note that existing applications may have unintentionally relied on this characteristic.

When the `disable escape` driver option is set, UNIFACE does not generate the ESCAPE clause. This option is supported solely to allow a simple migration to the U3.*x* driver, and its use may be discontinued in a future version of UNIFACE. It is recommended that you review and modify all applications that use ORACLE wildcard characters in UNIFACE retrieve profiles.

See section 4.3.1 *Retrieve profiles* for a detailed description of the ESCAPE clause, and for some examples on how it is used by UNIFACE.

Special 3GL services for ORACLE

There can be incompatibilities when user-defined 3GL accesses ORACLE:

- The U2.*x* ORA driver provided some undocumented features to allow user-defined 3GL to access ORACLE on the same logon path as the ORA driver using the Pro*C precompiler interface. The services provided by the ORA driver for user-defined 3GL have been formalized. These 3GL services are described in a read me file provided with the U3.*x* driver.
- If the `support obsolete 3gl services` driver option is set, the U3.*x* driver supports some of 3GL services as provided by the U2.*x* driver. These are described in the read me file provided with the U3.*x* driver. This option is supported solely to allow a simple migration to the U3.*x* driver, and its use may be discontinued in a future version of UNIFACE. It is recommended that you upgrade the user-defined 3GL to the interface described in the read me file.

By default, the U3.*x* driver creates the first logon path to ORACLE with the Pro*C precompiler interface. This allows user-defined 3GL to access ORACLE on this connection using the precompiler interface. Although this is completely transparent for most applications, it may cause an incompatibility with existing user-defined 3GL which creates the default connection.

See the read me file provided with the U3.*x* driver for more information. If the `disable precompiler connect` driver option is set, UNIFACE only makes use of the OCI interface to create non-default connections, allowing user-defined 3GL to create the default connection.

Number of cursors opened

Statement caching and cursor management has been completely redesigned in the U3.*x* ORA driver. For this reason, you must reconsider the number of cursors you allow UNIFACE to open. The $MAXCURSORSORA assignment supported in the U2.*x* ORA/ORT driver must be replaced by the open cursors driver option in the U3.*x* ORA driver.

The U3.*x* driver no longer supports $MAXCURSORSORA. When it is encountered, the ORA driver ignores it.

The $MAXCURSORSORA assignment is replaced by the open cursors *number* driver option in the USYS$ORA_PARAMS assignment.

Note: The statement caching and cursor management in the U3.x ORA driver have been redesigned; the open cursors driver option is not interpreted in the same way as the $MAXCURSORSORA assignment was in the U2.x driver. You should not assign the value that was previously assigned to $MAXCURSORSORA to open cursors. See section 4.6.1 Cursors and statement caching for more information on the open cursors driver option.

12.1.5 Review and modify assignment files

The U2.1 ORA/ORT and the U3.*x* driver are incompatible with respect to assignment files. Existing assignment files must be reviewed and modified for use with the U3.*x* driver. Furthermore, it is not possible for the U2.1 and the U3.*x* drivers to read the same USYS$ORA_PARAMS assignment. If the U2.1 and the U3.*x* drivers are both used on the same platform, you must arrange for the applications to read USYS$ORA_PARAMS from different assignment files. See the *UNIFACE Reference Manual* for more information on assignment files.

Review all assignment files which are read by your applications with the ORA driver. Check both the global assignment files (**usys.asn** and **psys.asn** in the UNIFACE installation directory), and the local assignment files, including **idf.asn** and **psv.asn**.

The assignment file incompatibilities are:

- The USYS$ORA_PARAMS assignment supports many new options, and old options are renamed. See chapter 10 *Overview of driver options (USYS$ORA_PARAMS)* for a complete overview of the driver options for the U3.*x* driver.
- The $MAXCURSORSORA assignment is ignored by the U3.*x* driver, and is replaced by the open cursors option in the USYS$ORA_PARAMS assignment.
- The U2.1 ORA/ORT driver requires a user name and password which you enter when logging on to ORACLE. The U3.*x* driver, on the other hand, allows ORACLE automatic logons (without a user name and password). This causes the application to behave differently. For example, when no logon path information is provided, the U2.1 driver displays the DBMS Logon form, whereas UNIFACE with the U3.*x* driver attempts an automatic logon. This incompatibility can be overcome by using question marks in the logon path specification.
- The U2.*x* ORA/ORT driver allowed the database field in the logon path specification to be preceded by the at (@) sign. This is not allowed by UNIFACE with the U3.*x* driver.

Table 12-1 describes how to convert existing assignments used with the U2.1 driver to assignments suitable for the U3.*x* driver when the two versions of the driver have to access the same database:

Table 12-1 How to change U2.1 driver assignments to U3.x driver assignments.

U2.1 ORA/ORT	U3.*x*
$maxcursorsora = *n*	USYS$ORA_PARAMS = open cursors *n*
No USYS$ORA_PARAMS	USYS$ORA_PARAMS = u2 default mapping
USYS$ORA_PARAMS = no_check	USYS$ORA_PARAMS = u2 default mapping, disable checks
USYS$ORA_PARAMS = enhanced_mapping	USYS$ORA_PARAMS = u2 enhanced_mapping
USYS$ORA_PARAMS = enhanced_mapping, no_check	USYS$ORA_PARAMS = u2 enhanced_mapping, disable checks
USYS$ORA_PARAMS = enhanced_mapping_2	USYS$ORA_PARAMS = u2 enhanced_mapping_2
USYS$ORA_PARAMS = enhanced_mapping_2, no_check	USYS$ORA_PARAMS = u2 enhanced_mapping_2, disable checks
No $ORA assignment or $ORA = ORA:||	$ORA = ORA:|?|?
$ORA = ORA:@t:*host*:*sid*|*user*|*password*	$ORA = ORA:t:*host*:*sid*|*user*|*password*

12.2 Migration from the U3.x to the U4.0 or U5.0 driver

Application models and assignment files which work with the U3.*x* drivers will work with the U4.0 or U5.0 driver. However, the following are incompatibilities between the U3.*x* drivers and the U4.0 and U5.0 drivers:

- Rowid
- Converting data with segmented fields
- Accessing ORACLE from 3GL

12.2.1 Converting data with segmented fields

When a new entity containing a segmented field is created, the U3.*x*, U4.0, and U5.0 drivers take the same user input. However, the U4.0 and U5.0 drivers do not limit you to one segmented field per entity, since this restriction has been removed in ORACLE8.

If an application contains one or more segmented fields, the U4.0 and U5.0 drivers create tables in which these are held as CLOB or BLOB data types. When converting data, the driver expects existing tables to hold these fields as CLOB or BLOB data types. However, the U3.*x* driver creates tables in which segmented fields are held as Long or Long Raw data types and expects existing tables to conform to this. Therefore, the tables corresponding to any existing entities which contain segmented fields must be converted.

The conversion process for tables with segmented fields is very similar to the process described in section 11.1 *The conversion procedure*, however, several extra steps are added. See section 11.1 *The conversion procedure* to learn how to set up your new U4.0 or U5.0 environment alongside your old environment and back up your data.

Complete the following steps to carry out the conversion procedure:

1. Identify the tables to be converted. These may include tables in the Application Objects Repository. See the *UNIFACE Reference Manual* for more information.
2. Backup all tables which need to be converted. You may consider this redundant, as the conversion procedure does not destroy old data, and the export to TRX format is a backup mechanism in itself. It is recommended to use this backup only as an extra safety precaution.
3. Switch to the old environment.
4. If the Repository is stored in ORACLE, use the Export Repository Objects form by selecting Tools–>Export to export the Repository to TRX.
5. Perform `analyze model` on all application models describing the data you want to convert. You must do this before exporting the data.
6. Export all tables which need to be converted, except for UOBJ and the tables from the Repository. To do this, use the Deployment–>Database Utilities–>Convert Data form, or the `/cpy` command line switch. For example, to export the `DEPT.PERSONNEL` table, you must enter the following on the command line:

   ```
   /cpy ora:dept.personnel trx:
   ```

7. Using SQL*Plus, identify the tables and packages associated with the entities to be changed. The tables normally have the same name as the entities and the package names are the entity names with '$U' appended. This may be confirmed by entering 'select object_name, object_type from user_objects;'. This instruction will also list constraints. These are automatically dropped when you drop a table.
8. Using SQL*Plus, drop the tables and associated packages. This is done by issuing the instructions '`drop package` <*package name*>;' and '`drop table` <*table name*>'.
9. Switch to the new environment.
10. Create the Repository in ORACLE, using the SQL scripts provided in the distribution kit. See section 2.1 *SQL scripts for creating the Application Objects Repository* for more information.
11. If the Repository was exported, as in step 4, use the Tools–>Import form to import the Repository from TRX. Run `/con` and `/all` on the command line to analyze the application models and to compile all global objects. See the *UNIFACE Reference Manual*, for more information on these command line switches.

12. Perform `analyze model` on all application models describing the tables you want to re-create. You must do this before using the Create Table utility.
13. Use the Create Table utility to generate SQL scripts which will create your databases. Note that a number of driver options affect the behavior of the Create Table utility with the ORA driver. See section 2.2.1 *Create Table utility*, for more information. Use one of the ORACLE utilities SQL*Plus or SQL*DBA to execute the generated script.
14. Import the tables that you exported in step 6. To do this, use the Deployment–>Database Utilities–>Convert Data form, or the `/cpy` command line switch. For example, to import the `dept.personnel` table, you must enter the following on the command line:

    ```
    /cpy trx:dept.trx ora:
    ```

15. Depending on how UOBJ was used, you may need to recompile global definitions using `/all` on the command line. See the *UNIFACE Reference Manual* for more information.
16. Rebuild and test your applications in the new environment. Note that installing a new ORA driver version may require that you relink all applications which use the driver. This depends on your operating system.

12.2.2 Accessing ORACLE from 3GL

For more information, see chapter 13 *Accessing ORACLE from 3GL*.

Chapter 13 Accessing ORACLE from 3GL

13.1 U3.x service functions

This section describes the use of the 3GL service functions provided by the U3.*x* ORACLE database drivers for use by 3GL routines which access ORACLE.

13.1.1 U3.*x* ORA versus U2.0 ORA

The ORACLE database drivers provide a number of services for user-defined 3GL to access ORACLE in cooperation with the driver. With the U3.3 driver for ORACLE7.3, these services are formalized and extended. The U3.3 driver supports all 3GL services offered by the U2.*x* drivers, but some services are only available when the option `support obsolete 3gl services` is set *on* in the `USYS$ORA_PARAMS` assignment setting.

Note: The `support obsolete 3gl services` option is supported only to facilitate migration from a U2.x to the U3.x driver, and it may be discontinued in a future version of UNIFACE. It is recommended you upgrade your 3GL to the superior services provided by a U3.x driver.

13.1.2 Pro*C versus OCI

It is possible to use the Pro*C precompiler interface, or the ORACLE Call Interface (OCI), or both in the same application.

13.1.3 Same logon path versus independent logon

Your 3GL functions can access ORACLE either on the same logon path as the ORA driver, or on an independent logon path (ORACLE connection):

- When you access ORACLE on the same logon path as your ORA driver, the database operations performed by your 3GL routines are in the same transaction as those of the ORA driver.
- When you open an independent concurrent connection, the database operations performed by user-defined 3GL and those of the ORA driver are in two independent transactions.

Note: On some systems, ORACLE does not support multiple concurrent connections, for example, single-user ORACLE under MS-DOS. Check your ORACLE documentation for more information.

13.1.4 First logon path

When you use your own 3GL routines with ORACLE, you must keep in mind the distinction between the first logon path to ORACLE, and subsequent concurrent logon paths. The first logon path is defined as:

- The first path opened after the application starts or
- The first logon path to ORACLE opened after the previous first logon path was closed

Note that this means it is possible to have a situation where there is no first logon path open. For example, consider the following sequence of events:

- A logon path to ORACLE is opened at 4GL level. This is the first logon path.
- A second concurrent logon path to ORACLE is opened via 4GL or 3GL.
- The first logon path is closed at the 4GL level. This means there is no longer a *first* logon path, even though a logon path is still open.
- A third logon path to ORACLE is opened. This is now considered the first logon path.

Only the first logon path of the ORA driver is accessible by user-defined 3GL.

13.1.5 Requirement for sharing a connection with the ORA driver

When your 3GL routines access ORACLE on the same connection as the ORA driver, you must meet the following requirements:

- Do *not* disconnect from ORACLE in your 3GL routines; instead, close the logon path to ORACLE at the 4GL level.
- Do *not* `commit` or `rollback` in user-defined 3GL; instead, `commit` and `rollback` on the logon path at the 4GL level.
- Ensure that the sum of the following items does not exceed the maximum number of cursors allowed by the ORACLE server:
  - The number of cursors opened by the ORA driver
  - The number of cursors opened by user-defined 3GL

  You may want to increase the OPEN_CURSORS initialization parameter of ORACLE. The maximum number of cursors opened by the U2.0 ORA driver is controlled by the `$MAXCURSORSORA` assignment setting. The maximum number of cursors opened by a U3.*x* ORA driver is controlled by the `open cursors` option of the `USYS$ORA_PARAMS` assignment setting.
- In most cases, user-defined 3GL can access the connection only after the ORA driver has successfully created the first logon path to ORACLE. The technique for the ORA driver, described in section 13.2 *Using a U3.x ORA driver*, requires that your 3GL routine creates the connection before the ORA driver logs on. In both cases, coordination of the order of events at the 3GL level and at the 4GL level may be required.
- Do not attempt to access the connection when the first logon path no longer exists.
- Your 3GL routines are responsible for closing the ORACLE cursors they have opened before the logon path is closed.

13.2 Using a U3.*x* ORA driver

Note: When you are using a U3.x driver, the driver must establish the first logon path to ORACLE. If necessary, force the driver to create the first logon path by opening the path at the 4GL level.

By default, a U3.*x* driver creates the first logon path to ORACLE using the Pro*C precompiler interface. The first logon path is associated with the default connection; that is, it is created with a `CONNECT` statement without an `AT` clause.

13.2.1 Accessing the driver's connection with the Pro*C interface

To access ORACLE on the same connection as the ORA driver using the Pro*C precompiler interface, your 3GL routines can access the first logon path of the ORA driver simply by executing SQL statements without an `AT` clause.

Do not set the ORA driver option `disable precompiler connect`, since this makes it impossible to access ORACLE on the same connection as the ORA driver using the precompiler interface.

13.2.2 Accessing the driver's connection with the OCI interface

To access the driver's connection to ORACLE, your 3GL routine must obtain the address of the LDA of the connection. You can use the 3GL service functions `UGETULDA` and `UGETUOPENFLAG` for this purpose. See the *3GL Interface Manual* for more information on these functions.

You must use the address of the LDA returned by `UGETULDA` in subsequent calls to the OCI routine `open()` to open cursors. Do not make a private copy of the LDA, as this will confuse ORACLE.

13.2.3 Accessing an independent connection with the Pro*C interface

User-defined 3GL can create either the precompiler default connection, or a nondefault connection. The default connection allows you to issue embedded SQL without explicitly identifying a connection.

To create the precompiler default connection, set the ORA driver option `disable precompiler connect` with the assignment setting `USYS$ORA_PARAMS`. If you do not set this option, the ORA driver creates its first logon path with the precompiler interface, and uses the default connection for this purpose. This can cause obscure ORACLE errors to occur either in your 3GL routines or in the ORA driver, since it cannot create more than one default connection. With the `disable precompiler connect` option set, the ORA driver uses the OCI routine `orlon()` to create an independent connection for the first logon path.

A default connection is made by a `CONNECT` statement that has no `AT` clause. For example:

`EXEC SQL CONNECT :`*user_password* `USING :`*database*`;`

The `USING` clause is optional. Refer to your ORACLE precompiler documentation for more information.

Alternatively, your 3GL can make one or more non-default connections using a `CONNECT` statement *with* an `AT` clause. Use an `AT` clause with SQL statements which must be executed on this connection. When your 3GL routine makes a non-default connection, you do not need to set the driver option `disable precompiler connect`. Refer to your ORACLE precompiler documentation for more information on the use of the `AT` clause.

13.2.4 Accessing an independent connection with the OCI interface

To create an independent connection, define a Host Data Area (HDA) and a Logon Data Area (LDA) and call the OCI function `orlon()`. Do not use the OCI routine `olon()` to do this.

13.2.5 Example using U3.*x* ORA driver

The following example (which is not a complete program) illustrates the use of the functions UGETULDA and UGETUOPENFLAG with the U3.*x* ORA driver. (These functions are described in the *3GL Interface Manual*.)

```
#include "oratypes.h"   /* The location of ORACLE 7.0 include files for OCI */
#include "ocidfn.h"     /* applications is platform specific. Refer to your */
#include "ociapr.h"     /* ORACLE documentation for more information. */

long my3gl( void )
{
    /* Declare the service functions of the ORACLE driver. */
    unsigned char *UGETULDA();
    unsigned char *UGETUOPENFLAG();

    unsigned char OpenFlag;
    unsigned char *LogonDataArea;

    sword ReturnStatus;


    /* Define a cursor. */
    struct cda_def Cursor;

    /* Dereference address returned by UGETUOPENFLAG() to get value of flag. */
    OpenFlag = *UGETUOPENFLAG();

    if ( OpenFlag != '\0' )
    {
       /* Get pointer to the ORACLE Logon Data Area. */
       LogonDataArea = UGETULDA();

       ReturnStatus = oopen( &Cursor, (struct lda_def *)LogonDataArea,
            (text *)0, (sword)-1, (sword)-1, (text *)0,
            (sword)-1 );
       if ( ReturnStatus != 0 )
       {  /* An abnormal error has occurred. Print an error message,
             including the return code in the Cursor Data Area
             (element Cursor.rc).
          */
       }
       /* Continue normal processing using the cursor. */
    }
    else
    {  /* An error has occurred: the first logon path is not open. Check
          your 4GL coding.
       */
    }
}
```

13.3 Using a U4.0 or U5.0 driver

The U4.0 and U5.0 drivers do not support Pro*C logon to the same path because this facility is not available using the ORACLE8 OCI functions. The UGETULDA and UGETUOPENFLAG functions are available for users who want to write functions which use OCI calls. In addition, two functions named URETURNULDA and UGETUSVCCTX are available. URETURNULDA must be used wherever UGETULDA is used. UGETUSVCCTX returns the service context for use in functions which use ORACLE8 OCI calls.

UGETUOPENFLAG can be used whether your function uses ORACLE7 or ORACLE8 OCI calls. The flag indicates whether UNIFACE has successfully logged on. The logon uses ORACLE8 functions to initialize handles rather than the LDA.

Where ORACLE7 OCI functions are to be used, the service context handle is converted into an LDA for return by UGETULDA. At the end of the user-written function which calls UGETULDA, you must call URETURNULDA, which converts the LDA back into a service context.

The synopsis of URETURNULDA is as follows:

```
void URETURNULDA (unsigned char * LogonDataArea);
```

Where ORACLE8 OCI functions are to be used, UGETUSVCCTX is used to get the address of the service context handle and UGETUENV is used to get the environment handle. The synopses are:

```
OCISvcCtx *UGETUSVCCTX(void);

OCIEnv *UGETUENV(void);
```

Chapter 14 Error messages

ORACLE error numbers and driver error numbers are returned in `$dberror`. ORACLE error numbers are positive, although negative error numbers may occur in some ORACLE environments such as PL/SQL. Driver error numbers are negative. When an ORACLE error or a driver error has occurred, a descriptive error message is available in the message frame.

Both the ORA driver and PL/SQL accumulate error messages. This means that when an error occurs, not only is the original driver or ORACLE error number returned in `$dberror`, but you may also see multiple error messages in the message frame. The most important driver errors which can occur are listed in table 14-1. ORA driver internal errors are not listed.

Table 14-1 Errors generated by the ORA driver and their messages. ***part 1 of 3***

$dberror value	Message
-4	Dynamic memory allocation failed.
-7	Value out of range: *value*.
-8	Syntax error or illegal value in USYS$ORA_PARAMS assignment.
-9	Invalid combination of driver options in USYS$ORA_PARAMS assignment.
-11	User-defined 3GL set uopenflag, but Logon Data Area (ulda) is invalid.
-13	Maximum number of concurrent logon paths to ORACLE exceeded.
-15	User-defined 3GL set uopenflag, but obsolete 3GL services are not enabled.

Table 14-1 Errors generated by the ORA driver and their messages. ***part 2 of 3***

$dberror value	Message
-16	User-defined 3GL set uopenflag, but the driver is already logged on.
-19	An element in the logon information string is too long.
-20	Syntax error in logon information string.
-23	Request for user name/password (Info mode 1) failed.
-24	User name without password, or password without user name specified.
-25	Row does not exist.
-27	Selected data too large for SQL Workbench (maximum length is 8K).
-30	Zero or more than one row(s) were deleted.
-31	Zero or more than one row(s) were updated.
-47	Package created with compilation errors. Query the ORACLE data dictionary view USER_ERRORS or DBA_ERRORS for the error messages.
-51	Generated SQL statement too large for internal buffer.
-54	Storage required for I/O buffer exceeds system limits. Reduce array size.
-55	The table would contain more than one LONG or LONG RAW column.
-56	The number of columns (plus one for the ROWID) exceeds ORACLE's maximum number of select list items.
-60	Number fetched from ORACLE exceeds size constraint of packing code.
-62	Data was truncated on fetch.
-66	One row expected, but zero or more than one row fetched.
-68	selectdb Proc statement is not supported on segmented fields. Field name: *field_name*.
-75	Actual length of LONG or LONG RAW data exceeds platform specific memory limitation.
-79	No SQL statement to process.

Table 14-1 Errors generated by the ORA driver and their messages. ***part 3 of 3***

$dberror value	Message
-80	Column has incorrect ORACLE storage format: Table *table_name*, Column *column_name*, expected storage format *storage format*, actual storage format is *storage format*.
-81	No more cursors available for statement processing. Increase value of 'open cursors' in USYS$ORA_PARAMS.
-82	Package does not exist, or is inaccessible: *package name*.
-83	Major version number of package in the database not acceptable: Package *package_name*, major version of package in database is *version_number*, current major package version number of ORA driver is *version_number*.
-86	Obsolete syntax in logon path specification: '@' preceding database.
-87	Incorrect number format.
-88	Invalid WHERE clause expression requested (check Proc u_where).
-89	odescr() OCI routine returned invalid data type code: Data type returned is not the correct type.
-90	Relationships in ORACLE not supported with a u2 packing code mapping