jEDI Development Kit User Guide

© 2006, 2016 Zumasys, Inc.

Contents

Documentation Conventions 1

INTRODUCTION 3

ORGANISATION OF THIS MANUAL 3

CONFIGURATION 4

jEDIdrivers.ini global parameters 4

MAPPING FROM DYNAMIC ARRAY TO RDBMS 5

THE MAPPING SCHEMA – CSV - DEFINED 5

FULLY EXPANDED EXPLAINED 7

FullyExpanded = 0 7

FullyExpanded = 1 8

FullyExpanded = 2 8

GENERATING THE MAPPING SCHEMA - JCREATECSV 10

jCreateCSV sequence of events 10

Accepted dictionary formats 11

Datetime fields 13

VALIDATING THE SCHEMA – JCHECKSCHEMA 14

What happens if I try to write invalid data? 14

If there are *EXCEPTIONS* how does a SELECT work? 16

How does Exception handling affect performance? 16

What if I don't want the *EXCEPTIONS* table? 16

I have a parameter file where records are of different formats. How can I store that in an RDBMS? 16

How can I see what the driver is doing? 17

SAMPLE CSV FILE 17

APPENDIX A - PLSORA DRIVER 20

INTRODUCTION 20

ORACLE REQUIREMENTS 20

Server Requirements 20

Client Requirements 20

ORACLE DATABASE CONFIGURATION 21
ORACLE DATABASE OBJECTS 21
Tablespaces 21
User (createuser.sql) 21
Exception Table (CreatejBASEpkg.sql) 22
Packages and Functions (createfunctions.sql) 22
DRIVER INSTALLATION 22
DRIVER CONFIGURATION 23
USING THE DRIVER 24
CREATE-FILE 24

APPENDIX B - DB2EXP DRIVER 29

INTRODUCTION 29
DB2 REQUIREMENTS 29
Server Requirements 29
Client Requirements 29
DB2 CONFIGURATION 30
DB2 Database Objects 30
DRIVER INSTALLATION 31
DRIVER CONFIGURATION 31
USING THE DRIVER 33
CREATE-FILE 33

APPENDIX C - OLESQL DRIVER 37

INTRODUCTION 37
SQL SERVER REQUIREMENTS 37
Server Requirements 37
Client Requirements 37
SQL SERVER CONFIGURATION 37
SQL SERVER DATABASE OBJECTS 38
User 38
DatabaseObjects (CreateExceptions.sql) 38
INSTALLATION 38
CONFIGURATION 39
jEDIdrivers Configuration 39
USING THE DRIVER 39
CREATE-FILE 40

APPENDIX C - FREQUENTLY ASKED QUESTIONS 44

How does jBASE find the driver? 44
How does jBASE use the driver? 44
What happens if I try to write invalid data? 45
How does a SELECT work if there are *EXCEPTIONS*? 45
How does Exception handling affect performance? 46
What if I don't want the *EXCEPTIONS* table? 46
I have a parameter file where records are of different formats. How can I store that ? 46
How can I see what the driver is doing? 46
What stored procedures does Oracle use? 47
What stored procedures does SQL server use? 47
What happens when the driver reads? 48
What happens when the driver writes? 50
What is the locking strategy? 51
When do records get committed 51
How are transactions implemented 52
When are database connections opened and closed 52
How is I18N UTF-8 data handled? 52
What does the jstat command show me? 52
What do the following errors mean? 53

Documentation Conventions

This manual uses the following conventions:

Convention	Usage
BOLD	In syntax, bold indicates commands, function names, and options. In text, bold indicates keys to press, function names, menu selections, and MS-DOS commands.
UPPERCASE	In syntax, uppercase indicates JBASE commands, keywords, and options; BASIC statements and functions; and SQL statements and keywords. In text, uppercase also indicates JBASE identifiers such as filenames, account names, schema names, and Windows NT filenames and pathnames.
UPPERCASE *Italic*	In syntax, italic indicates information that you supply. In text, italic also indicates UNIX commands and options, filenames, and pathnames.
Courier	Courier indicates examples of source code and system output.
Courier Bold	**Courier Bold** In examples, courier bold indicates characters that the user types or keys (for example, <Return>).
[]	Brackets enclose optional items. Do not type the brackets unless indicated.
{ }	Braces enclose nonoptional items from which you must select at least one. Do not type the braces.
ItemA | itemB	A vertical bar separating items indicates that you can choose only one item. Do not type the vertical bar.
. . .	Three periods indicate that more of the same type of item can optionally follow.
⇒	A right arrow between menu options indicates you should choose each option in sequence. For example, "Choose **File ⇒Exit**" means you should choose **File** from the menu bar, and then choose **Exit** from the File pull-down menu.

Syntax definitions and examples are indented for ease in reading.

All punctuation marks included in the syntax—for example, commas, parentheses, or quotation marks—are required unless otherwise indicated.

Syntax lines that do not fit on one line in this manual are continued on subsequent lines. The continuation lines are indented. When entering syntax, type the entire syntax entry, including the continuation lines, on the same input line.

INTRODUCTION

The jBASE database adheres to a similar model of the Reality/PICK database storing records as ‘dynamic arrays’ in ‘hashed’ files. You can view these records in a manner similar to SQL using jQL (jBASE’s version of the ENGLISH/ACCESS query language from Reality/PICK). However, as the jBASE query language is not (currently) ANSI SQL compliant there are those who feel the need to use a ‘relational’ database (typically referred to as RDBMS). With the development of the jEDI architecture in jBASE the ability to use any database to store records was made possible. The problem for most (if not all) jBASE users is mapping their existing record structure on to the RDBMS. The *jEDI Development Kit* (jDK not to be confused with JDK – Java Development Kit) attempts to automate this as much as possible. There are several ‘drivers’ (shared objects), which work in conjunction with the jDK. These drivers work with their own specific RDBMS but rely on the jDK to generate the mapping between the dynamic array – on the jBASE side – and the columns in tables – on the RDBMS side.

ORGANISATION OF THIS MANUAL

This manual is split into two main sections. The first section provides generic information that is relevant to all jEDI Development Kits. The latter section provides detailed information specific to particular jDK drivers. It is recommended for new users of the jDK to read the first section before referring to the section dedicated to the particular driver of interest.

CONFIGURATION

The jDK driver suite relies, to some extent, on an initialisation file for critical parameters; called *jEDIdrivers.ini* its functions are similar to a "profile" on Unix operating systems. In fact the file is expected to be placed in **/etc** for global settings and if placed in *$HOME* it is named **.jEDIdrivers.ini**. On Windows the file is always **jEDIdrivers.ini** but can exist in *%SYSTEMROOT%* or *%HOME%.*

jEDIdrivers.ini global parameters

Below is an example of the non-driver parameters with a description
(***note the parameters ARE case sensitive)***:

[General]

JRLAlock = 1	This specifies that jBASE will handle the locking of records (currently the jDK drivers do not handle locking at the RDBMS level).
PasswdsEncrypted = 0	Specifies whether passwords for RDBMS logins are encrypted
FullyExpanded = 2	Controls how multi-values are stored across primary and secondary tables (more explanation later in this document).
TblSep = __	Literal separator characters between primary and secondary tables (more explanation in this document).
CSVdir = /home/myuser/csv	Specify the directory where **csv** definitions are stored for controlling the mapping between attributes and columns
JdkHome = /home/myuser/jDK	This is the top directory tree where other jDK parameters/scripts/etc. are located.
JdkTempPath = /temp	A temporary directory where output of jDK tools are written during file creation

.

MAPPING FROM DYNAMIC ARRAY TO RDBMS

Generally, attributes in a dynamic array (*record*) are mapped to an individual column in a table (*row*).

The Mapping schema – CSV - defined

The mapping for the jDK drivers is driven by *csv* (comma separated value) files; the fields are:

ColumnName	Must be a valid *SQL* column name
Attribute	Usually a single numeric (with the exception of DT types explained later in this document)
Width	Mandatory integer for character type fields
Type	A{N} - Varchar C - Char D - Date T - Time DT - DateTime UTC - Universal Time Conversion N*d* - Numeric (where *d* is optional number of decimals)
Group Association	Optional group name for repeating groups. For sub-values the required format is: {multi-value name|sub-value name}. This name will be appended to the primary table name (with a separation character of # unless ***TblSep*** is defined in **jEDIdrivers.ini**) when creating the associated table.
Group Controlling Flag	Y or 1 can be entered or left blank. This should be entered against the field which will

	always have a value when more than one field has the same *Group Association*.
Not Null Flag	Y or 1 for Not Null (i.e. mandatory) or N or 0 (or blank) for non-mandatory
Positional multi-value	For multi-value (or sub-value) fields, which are non-repeating you can enter the multi-value (or sub-value if within a multi-value group) position.
Compound Field Literal	For fields like date*time you can enter the literal separation (e.g. *). The literal is applied to the field whose value appears before the literal. Multiple compounds can be defined. That last field in the compound would not have a literal (unless the value actually ends in a constant literal). This literal value does not appear in the RDBMS. The driver adds/removes it when reading/writing from/to the RDBMS.
Foreign Source	Use this field to facilitate foreign key relationships at the RDBMS level. Format must be *table_name.column_name* and *table_name* must exist in the RDBMS.

There must be at least two lines, the first pointing to attribute 0 and at least one line pointing to an attribute other than 0. One exception is an **empty** csv file, which implies a 2-column table where the entire record (dynamic array) is stored in a *BLOB* type column.

Fully Expanded Explained

This parameter (from *jEDIdrivers.ini*) has three settings: 0, 1 and 2. It affects the way the first multi-value (or sub-value) is handled on the target RDBMS when defining *Associations*. Because the hash file model can read and write whole records faster than most (if not all) RDBMS an attempt has been made to improve performance of a jEDI by reducing the number of *fetches* and *insert/update* operations.

EXAMPLE

Suppose you have a Customer file with a *Contacts* field, which can repeat. This would require two tables:

One for the primary *Customer* details and
Another for the repeating *Contacts* (ideally just a *key* to a Contact file/table).

When typically reading an existing customer you would *SELECT* the columns from the Customer file (table) and the Contacts file (table); Now suppose that virtually all the Customer records never have more than one contact (even though they have the facility to in the database). The READ operation always has to do two fetches (note, this is not necessarily true in the jDK driver suite due to the *VMC_column* behaviour previously explained) to bring back all the details.

Customer File

Name		
Address		
Contact # 1	Contact # 2	Contact # 3

FullyExpanded = 0

Using the above example the jDK forces the first multi-value to appear in the primary table (i.e. *Customer*). If the customer being read has only one contact the whole record is retrieved with a single fetch (the **VMC_Contacts** column would be '1' so the driver knows no more fetches are required). You can create *Views* (the jDK can automate some of this), which combine the first multi-value to the associated repeating table's values to make the associated set appear as one.
The disadvantage of this format is the reshuffling of data if the first multi-value is deleted. Currently the jDK drivers do not optimise updates therefore, it updates the whole record.

In the example below, there are three contacts associated with this customer record; the **VMC_Contacts** value would be 3.

Customer File

Name
Address
Contact # 1

Contacts File

Contact # 2	Contact # 3

FullyExpanded = 1

This is similar to zero except it repeats the first multi-value in both the primary table (i.e. *Customer*) and associated table (i.e. *Contacts*). The advantage of this is that a *view* is not required to query the repeating fields. The disadvantage is that if updates are made outside the jBASE environment the user must be aware that the first value must be updated in two tables (e.g. in this case *Customer* and *Contacts*).

Keeping with the previous example **VMC_Contacts** value would be 3.

Customer File

Name
Address
Contact # 1

Contacts File

Contact # 1	Contact # 2	Contact # 3

FullyExpanded = 2

This format is more like the traditional RDBMS way of handling repeating groups. All the repeating fields are stored in their respective table(s). There is still a *VMC_column* in the primary table for each associated group therefore the driver knows how many rows to expect.

Again, in the example below **VMC_Contacts** value would be 3.

Customer File

Name

Address

Contacts File

Contact # 1	Contact # 2	Contact # 3

GENERATING THE MAPPING SCHEMA - JCREATECSV

This command generates the comma-separated file (CSV) used for mapping the dynamic array to the columns in a relational table. It gets its information from an existing dictionary of a file in jBASE (note, this can be any file that satisfies an OPEN 'DICT' command in jBASIC); jBASE supports a variety of dictionary formats, as does jCreateCSV.

SYNTAX

```
jCreateCSV file_name target.csv {prefix_chars} {options}
```

options:	- D{*type*}	dictionary type build driver valid *types*: J for jDC, P or U for Prime/Unidata A option (appended to J) for A types only (i.e. do not include 'S' type dictionaries in the mapping).
- O		overwrite previous definition
prefix_chars:		Leading characters to strip from dictionary when creating column names in comma separated file this is useful in cases where a standard prefix has been used to key the dictionaries (many 4GLs do this) and you do not want this prefix used on the column names.

jCreateCSV sequence of events

First, it selects, reads and vets all the dictionaries (e.g. if the A option was used it ignores 'S' type dictionaries) and ignores duplicates. Sorting is in ascending attribute order. There **must** be at least one definition pointing to attribute 0 (i.e. the record key).

Sorting is in ascending attribute order and scanned for associated groupings. Finally, it writes a resulting CSV to the *CSVdir* (as defined in the *jEDIdrivers.ini*).

Accepted dictionary formats

The classic Reality/PICK style – ‘A’ (or ‘S’) – type dictionary describes individual attributes reasonably well. Unlike the ‘D’ type dictionary, it does not, support the definition of repeating groups (multi-value/sub-value fields). To accomplish this you must use the *jBASE Extended Dictionary*. These are extra attributes, which appear after the last ‘normal’ attribute of a Reality/PICK style dictionary, typically starting at attribute 30 (although this is user definable).

For the ‘D’ type approach you can also include ‘V’ or ‘I’ types to handle things like:

Compound fields – e.g. DATE*TIME
e.g. FIELD(@*defining_dict*, “*”, 2)
Position multi-values
e.g. EXTRACT(@RECORD, *attr*, *mv_pos*)

For ‘A’ (or ‘S’) type dictionaries you can achieve similar results:

Compound field example
…
<8> G1*1
…
Positional multi-values
<1> S
<2> 0
<8> Tfilename;Xmv_pos;;attr

In the case where the field type is not explicit (i.e. the *Extended Dictionary* is either missing or incomplete) the following rules apply:

Dictionary Property	Field type
Conversion field starts with D	D - DATETIME - where only the date is used - or DATE (some RDBMS platforms support Date type columns)
Conversion starts with MT	T - DATETIME - where only the time is used - or TIME (some RDBMS platforms support Time type columns)

Dictionary Property	Field type
Conversion of UTC	UTC - DATETIME
Conversion starts with MD or MR	N*d* - DECIMAL
Justification is R	N - NUMBER
Justification is L	C - CHAR Character Fixed Length
Other	A{N} - VARCHAR Character Variable Length

Here is an example of a generated CSV from dictionaries often found in the ***jDP*** samples (note the *Controlling* value against HARDWARE was added manually afterwards).

Column	Attr	Width	Type	Association	Controlling
ID,	0,	10,	AN		
FIRSTNAME,	1,	24,	AN		
LASTNAME,	2,	20,	AN		
ADDR1,	3,	21,	AN		
ADDR2,	4,	20,	AN		
CITY,	5,	11,	AN		
STATE,	6,	3,	AN		
ZIP,	7,	12,	AN		
HOMETEL,	8,	16,	AN		
WORKTEL,	9,	16,	AN		
EMAIL,	10,	25,	AN		
HARDWARE,	11,	15,	AN,	HARDWARE,	1
OS,	12,	15,	AN,	HARDWARE	
SYSTEMTYPE,	13,	24,	AN,	HARDWARE	
NUMUSERS,	14,	6,	AN,	HARDWARE	

This definition implies two tables:

Primary table (CUSTOMER) which holds all the fields with no *Association*

Multi-value association – *HARDWARE* – which generates the table CUSTOMER#HARDWARE

Datetime fields

As RDBMS databases typically have a *datetime* field whereas jBASE internally uses *date* and *time* separately, it is possible to make use of both portions by defining a DT type field in the *CSV*. This splits the date and time into separate entities on jBASE, which merges on the RDBMS side. It supports the following combinations:

Date and Time on separate attributes.
To achieve this specify the *Attribute* field as *date_attr|time_attr*.
Date and Time on the same attribute but specific multi-values.
Simply specify the DT type field with a single attribute.
Date and Time in the same field as a compound field.
Same as option 2 but specify a *SplitChar*.

Currently maintained at the *CSV* level

Here as an example of all three scenarios:

Column	Attr	Width	Type	Association	Controlling	NotNull	Position	SplitChar
CUST,	0,	10,	AN,	,	,	1,	,	
SPLITDT,	1|2,	,	DT,	,	,	1,	,	
MVDT,	3,	,	DT,	,	,	1,	,	
COMPDT,	4,	,	DT,	,	,	1,	,	*

The above csv has a key field of *CUST* and three Datetime fields

SPLITDT
The *date* will appear in attribute 1 and the *time* in attribute 2
MVDT
Will appear as *date*]*time* on attribute 3 (i.e. multi-values 1 and 2)
COMPDT
Will appear as *date***time* on attribute 4.

VALIDATING THE SCHEMA – JCHECKSCHEMA

Once you have generated the CSV run jCheckSchema against an existing file (typically a hashed file). This will highlight any problem areas (e.g. invalid ASCII characters in a numeric, date or time field) and automatically adjust lengths of alpha{-numeric} fields (as the RDBMS world uses fixed length fields for it's character fields).

Syntax

```
jCheckSchema file_name csv_definition {recordkey} {-options}
```

options:	i	interactive – this means that it will prompt for input where it normally make an adjustment
	q	quiet – no progress is displayed (normally a percentage of records processed is shown)
	rn	rounding factor where the length of character fields (A{N}) are rounded up to the next factor of n.
	R	report only – no adjustments are made to the CSV.

What happens if I try to write invalid data?

The CSV definition used when creating the file – and table(s) – determines what data types are valid and which fields are repeating. Unlike jBASE, it can only store values that are consistent with these data types. Some of the drivers have built-in logic for handling invalid records. The **TrapErrs** parameter in the jEDIdrivers.ini determines its use. Setting this variable to '1' ensures that all writes will succeed regardless of whether the record is consistent with the CSV structure or not. The way this is handled is via an ***EXCEPTIONS*** table (created as part of the *driver* installation). This table holds the following fields:

filename
item-id
update timestamp
program which performed the write
dynamic array record image

When a jBASE application tries to read back this record it first attempts to read from the intended table. If not found it looks in the ***EXCEPTIONS*** table and the application does not perceive any difference in logic flow.

It is possible to setup automatic emails to notify various parties when an exception occurs. Although this is outside the boundaries of jBASE, the database administrator can implement this feature without interfering with the workings of the driver.

If there are *EXCEPTIONS* how does a SELECT work?

A *SELECT* from jBASE initiates a *SELECT* from the primary table. If **TrapErrs** is set to '1', it includes a ***UNION ALL*** on the EXCEPTIONS table based on the current filename.

How does Exception handling affect performance?

The problem with relying on **TrapErrs** to catch your mistakes is that there is extra work needed by the driver to ensure data integrity. The following points highlight this:

> For every read where no record is found on the primary table, an additional SELECT is performed on the ***EXCEPTIONS*** table (the same goes for **deleting** records).
>
> For writes, the driver tries to detect if invalid data is being used, then calls the stored procedure to handle and update the table(s). If any update fails the driver has to then update the ***EXCEPTIONS*** table.
>
> If performing a write on a table with no repeating groups it needs to check if the record was previously in the ***EXCEPTIONS*** table and if so delete it.
>
> Any SELECT has to include the ***EXCEPTIONS*** table to ensure all records are retrieved.

What if I don't want the *EXCEPTIONS* table?

Set **TrapErrs** to 0. This means that an invalid write from jBASE will cause an error. The offending program can be coded with an ***ON ERROR*** clause and handle the error programmatically. If this is not done the application will see a write error message with:
(I)gnore, (R)etry, (Q)uit
options

I have a parameter file where records are of different formats.
How can I store that in an RDBMS?

If you use an empty CSV file as the definition for your table, it creates the records with a generic record_id and record image (i.e. the dynamic array); referred to (sometimes) as a 'BLOB'.

How can I see what the driver is doing?

There are two tracing methods: screen and logfile. First, you must set a trace level:
`%JEDI_`*`drivertype`*`_TRACE%`

For a basic trace of I/O set this environment variable to '1'. For a more detailed trace use '2' (more detail levels may be introduced later).
Next, you can optionally define:
`%JEDI_`*`drivertype`*`_LOG% to '1' and %JEDI_`*`drivertype`*`_LOGFILE% to the name of the file you want to store the trace information.`

You can also set:
`%JEDI_`*`drivertype`*`_DISPLAY% to '1' to display the log to the screen at the same time.`

NOTE: if you do not set %JEDI_*drivertype*_LOG% it assumes %JEDI_*drivertype*_DISPLAY%.

SAMPLE CSV FILE

`ZONE,0,10,C,,,1,,.`	CHAR(10). The first 3 lines represent a 3-part key: *ZONE*, *CUST* and *SEQ*. The "." and "#" that separate the fields are only seen from the jBASE point of view.
`CUST,0,10,C,,,1,,#`	CHAR(10). All key fields have the NotNull set (which is the default for key fields).
`SEQ,0,2,N,,,1`	NUMBER(2). The last field in the compound definition has no *SplitChar*.
`NAME,1,20,AN,,,1`	VARCHAR(20). Mandatory alphanumeric field on attribute 1.
`AREA_CODE,2,3,N,,,1,,-`	NUMBER(3). First part of a compound field on attribute 2 delimited by "-".
`PHONE_NUMBER,2,7,N,,,1`	NUMBER(7) on attribute 2 being the 2nd part of the compound field.
`STREET,3,50,AN,,,1,1`	VARCHAR(50) on attribute 3 multi-value 1,

	mandatory.
`TOWN,3,20,C,,,1,2`	CHAR(20) on attribute 3 multi-value 2, mandatory.
`ZIP,3,5,N,,,1,3`	NUMBER(5) on attribute 3 multi-value 3, mandatory.
`COUNTRY_CODE,3,5,C,,,1,4,,COUNTRY.CODE`	CHAR(5) on attribute 3 multi-value 4, mandatory with a foreign key constraint of *COUNTRY.CODE.*
`BALANCE,4,10,N2`	DECIMAL(10,2) on attribute 4.
`CONTACT,5,20,AN,CONTACTS,,1`	VARCHAR(20) on attribute 5. Repeating field in associated table **CUSTOMER#CONTACTS**.
`MACHINE,6,10,AN,HARDWARE,1,1`	VARCHAR(10) on attribute 6. Repeating field in associated table **CUSTOMER#HARDWARE**.
`BACKUPTIME,7,8,T,HARDWARE,,1`	{DATE}TIME on attribute 7. Repeating field in associated table **CUSTOMER#HARDWARE**.
`SOFTWARE,8,20,AN,HARDWARE|SOFTWARE,,1`	VARCHAR(20) on attribute 8. Repeating field in associated table **CUSTOMER#HARDWARE#SOFTWARE** (i.e. sub-valued within each multi-value for *HARDWARE*).
`SYSADMIN,9,20,AN,HARDWARE,,1,1`	VARCHAR(20) on attribute 9. First sub-value in associated group (i.e. positional sub-value within each multi-value for *HARDWARE*).
`NETWADMIN,9,20,AN,HARDWARE,,1,2`	VARCHAR(20) on attribute 9. Second sub-value in associated group.
`SUBVALUES,10,20,AN,ONE|MULTIVALUE,,1,1`	VARCHAR(20) on attribute 10. By placing a *Positional* value of 1 on a sub-value definition that means the sub-values are in a non-repeating multi-value. You could repeat this definition for *positions* 2, 3, etc. Each *multi-value* would have repeating sub-values.
`COMPDT,11,10,DT,,,1,,*`	DATETIME on attribute 11 will appear as *date*time*.
`MVDT,12,10,DT,,,1`	DATETIME on attribute 12, multi-valued as *date*]*time*
`SEPARATEDT,13|14,10,DT,,,1`	DATETIME where the *date* appears on attribute 13 and the *time* on attribute 14.

`LASTUPDATED,15,,UTC`	DATETIME in number of seconds in UTC format stored on attribute 15.
`CREATED,16,,D,,,1`	DATE{TIME} appearing on attribute 16.

NOTE: you do not have to define every attribute. Missing attributes will not be used which is useful when migrating an existing file and other attributes have become redundant.

APPENDIX A - PLSORA DRIVER

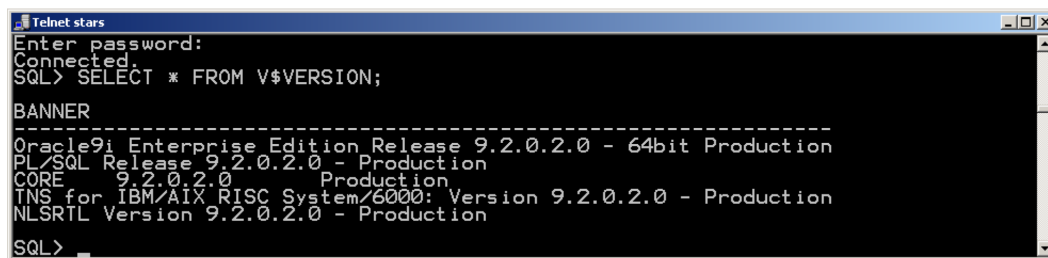
Introduction

The jDK PLSORA driver bridges the gap between jBASE and ORACLE with little or no intervention of the jBASE application developer, so that it is transparent for jBASE applications to access Oracle database. Moving the data from the traditional 'hash file' environment to the RDBMS (e.g. ORACLE) environment brings with it one intrinsic point of which a user should be aware. The 'multi-value' environment is different to the 'RDBMS' environment in the sense that the underlying management of data is different. From a high-level user perspective, they appear (and in fact are) both the same - "the application retrieves data from the database". The method by which the database drivers deliver and manipulate the data 'under the covers' are however very different. Put simply – performing a 'COUNT' operation in one manufacturers database will yield the same results as performing a 'COUNT' on the same data in a different manufactures database, but the method by which the count was generated will differ.

Oracle Requirements

Server Requirements

Oracle Server version must be 9I Release 2 version 9.2.0.2.0 Enterprise Edition or Standard Edition.
You must install the Oracle OCI and PL/SQL packages on the server.
To determine the Oracle version and edition, issue the command 'select * from v$version ' from an 'SQLPLUS' session (dba permissions may be required):



Client Requirements

The Oracle (database) server and jBASE server can reside on the same machine or on separate machines. If the Oracle (database) server and jBASE servers reside on separate machines, it will be necessary to install the Oracle Client software on the jBASE server and configure the Oracle Net Services to reference the database through an appropriate naming method (Local Naming, Oracle Names Service or Host Naming).

Oracle Database Configuration

It is recommended that you use a case sensitive database to enable the different treatment of record keys like ‘A’ and ‘a’.
The following parameters are recommended for the Oracle database:

Block Size – 4096

Character Set - AL32UTF8, UTF8 or WE8ISO8859P1

The following parameters need to be set in the oracle initialisation (spfile) file:

query_rewrite_enabled = TRUE

query_rewrite_integrity = TRUSTED

Oracle Database Objects

Create the following objects on the target database:

Tablespaces

- It is recommended that two separate tablespaces are created on the Oracle database
- DATA Tablespace eg. MYDATA (contains all the application Tables)
- INDEX Tablespace eg. MYINDEX (contains all the application Indexes)
- The tablespaces should be created with the ‘locally managed and auto-allocate’ options. Set the size of each tablespace to initially 1Gb (with auto-extend) for a typical application.
- Use any name for the tablespaces. The driver from the configuration file reads these names.

User (createuser.sql)

To run this you need to connect to a dba user (e.g. system)

At sqlplus @scripts/createuser <uid> <passwd> <dflt_tblspc>

The user should have an unlimited quota on the default tablespace.

On later versions of the PLSORA package the createuser.sql script also runs the following scripts automatically

Exception Table (CreatejBASEpkg.sql)

Should be run from sqlplus while logged in as the user the application will use.

The PLSORA driver may require an EXCEPTIONS table and accompanying stored procedures installed on the database.

You can find the 'CreatejBASEpkg.sql' file in the PLSORA driver 'zip/tar' file, which can be run from 'SQLPLUS' session by using an '@' character followed by the full path to the script.

Packages and Functions (createfunctions.sql)

Should be run from sqlplus while logged in as the user the application will use.

The PLSORA driver requires installation of a few functions and packages on the database (ANNDDL, FIELD, EXTRACTEX, TIMECONV and PLSDBG).

The 'createfunctions.sql' file can be found in the PLSORA driver 'zip/tar' file and can be run from an 'SQLPLUS' session by using an '@' character followed by the full path to the script.

Driver Installation

The major components of the PLSORA driver include:

libPLSORA.so and libPLSORA.so.el (Windows version libPLSORA.dll and libPLSORA.def)	The actual driver files. These files need to reside in the jBASE 'lib' directory ($JBCRELEASEDIR/lib) or an appropriate directory in the $JBCOBJECTLIST path.
jEDIdrivers.ini (On non-Windows platforms this would have a leading "." if placed in the user's $HOME)	Configuration file for holding database and parameter information.
jBuildORA	Program used to generate the stored procedures (PL/SQL scripts) on Oracle.
Templates	A directory holding skeleton stored procedures used by jBuildORA
jDK tool set (comprising of executables and libraries).	These programs should reside in the jBASE 'bin' directory ($JBCRELEASEDIR/bin) and jBASE 'lib' directory ($JBCRELEASEDIR/lib). Equivalent bin (included in $PATH) and lib (included in $JBCOBJECTLIST) will suffice.

Once the driver has been installed, modify the users' environment to allow the driver to operate. Set the following environment variables:

PATH	Set to include the '$JBCRELEASEDIR/bin' directory
JEDI_SOB_NOCLOSE	Set to the value '1' for correct driver operation

As a generic test, if the user can begin an 'sqlplus' session, then the PLSORA driver should operate correctly

Driver Configuration

Before the driver can operate, you must create the configuration file 'jEDIdrivers.ini'. This file physically holds the configuration parameters, which the driver reads.
Create the 'jEDIdrivers.ini' file in one of two places:

/etc or In the user's home directory ($HOME)

It is recommended that you create the file in the "/etc" directory. This allows multiple users access to the same configuration. It is possible to have both whereby the $HOME version will compliment and override settings from the /etc version.

The configuration for the PLSORA driver is controlled by a [PLSORA] section heading in jEDIdrivers.ini. The minimum requirements for connecting to the target Oracle database is:

user = <oracle_user>
passwd = <password>

If the PasswdsEncrypted=1 (either locally under [PLSORA] or globally under [General]) the <password> must be encrypted. Refer to the first part of this document for information on encrypting password.

Additionally you may need:

```
database = <ORACLE_SID>
```

{If ORACLE_SID is exported then this is not required}

Using the Driver

This section lists the commands that are specific to the PLSORA Driver or have extensions for the PLSORA Driver.

Definitions

Before you create a PLSORA type file, you must have a *CSV* definition to map each record from dynamic array to relational. For details on this, refer to the first section of this manual.

CREATE-FILE

Syntax

CREATE-FILE *filename* TYPE=PLSORA {TABLE=*tablename*} {CSV=*csvdefinition*} {EXISTING=YES}

tablename defaults to filename
csvdefinition defaults to tablename (this definition is read from the directory specified by the **CSVdir** parameter in the jEDIdrivers.ini file).
Any '.' characters in the filename or columns will be converted to a '_' character in the database

Example:

```
CREATE-FILE PLSCUSTOMER TYPE=PLSORA CSV=JCUSTOMER
```

(Executed from a telnet session)

```
[clee@bench csv]$ CREATE-FILE PLSCUSTOMER TYPE=PLSORA CSV=JCUSTOMER
[ 417 ] File PLSCUSTOMER]D created , type = PLSORA
[ 417 ] File PLSCUSTOMER created , type = PLSORA
Disconnected from jBASE at 10:53:54  12 JAN 2005
[clee@bench csv]$ ls
BL_ACCT]D   BL_ACCT_J4.csv  CARS.csv    EQUIPMENT.csv  EQUIPMENT_MASTER.csv  ERIC.csv       PLSCUSTOMER
BL_ACCT_J4  BL_ACCT_J4]D    ENTCDE.csv  EQUIPMENT]D    EQUIPMENT_MASTER]D    JCUSTOMER.csv  PLSCUSTOMER]D
[clee@bench csv]$
```

NOTE:

That although the dictionary portion of the above file – PLSCUSTOMER]D – is displayed as type = PLSORA it is in fact a regular j4 hashed file. You could if you wanted to create them separately thus:

```
CREATE-FILE DICT PLSCUSTOMER 1
CREATE-FILE DATA PLSCUSTOMER TYPE=PLSORA CSV=JCUSTOMER
```

This create-file command does three things:

Creates a dictionary for the file (unless DATA has been specified)

Generates a PL/SQL script to run against Oracle by executing the ***jBuildORA*** command against the csv file: JCUSTOMER.csv

Runs the script, which creates the necessary table(s) and stored procedures.

Writes out a *stub* file – PLSCUSTOMER – to the current working directory which will look like this:

JBC__SOB JediInitPLSORA PLSCUSTOMER

In the above example our CSV looked like this:

Column	Attr	Width	Type	Association	Controlling
ID,	0,	10,	AN		
FIRSTNAME,	1,	24,	AN		
LASTNAME,	2,	20,	AN		
ADDR1,	3,	21,	AN		
ADDR2,	4,	20,	AN		
CITY,	5,	11,	AN		
STATE,	6,	3,	AN		
ZIP,	7,	12,	AN		
HOMETEL,	8,	16,	AN		
WORKTEL,	9,	16,	AN		
EMAIL,	10,	25,	AN		
HARDWARE,	11,	15,	AN,	HARDWARE,	1
OS,	12,	15,	AN,	HARDWARE	
SYSTEMTYPE,	13,	24,	AN,	HARDWARE	
NUMUSERS,	14,	6,	AN,	HARDWARE	

```
LASTMAINT,     15,   11,    DT,   HARDWARE|UPDATED,   1
BALANCE,       16,   12,    N2
```

This definition implies three tables:

Primary table (PLSCUSTOMER) which holds all the fields with no *Association*

```
SQL> desc PLSCUSTOMER
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ID                                        NOT NULL VARCHAR2(10)
 FIRSTNAME                                          VARCHAR2(24)
 LASTNAME                                           VARCHAR2(20)
 ADDR1                                              VARCHAR2(21)
 ADDR2                                              VARCHAR2(20)
 CITY                                               VARCHAR2(11)
 STATE                                              VARCHAR2(3)
 ZIP                                                VARCHAR2(12)
 HOMETEL                                            VARCHAR2(16)
 WORKTEL                                            VARCHAR2(16)
 EMAIL                                              VARCHAR2(25)
 HARDWARE                                           VARCHAR2(15)
 OS                                                 VARCHAR2(15)
 SYSTEMTYPE                                         VARCHAR2(24)
 NUMUSERS                                           VARCHAR2(6)
 LASTMAINT                                          VARCHAR2(11)
 BALANCE                                            NUMBER(12,2)
 VMC_HARDWARE                                       NUMBER
 VMC_HARDWARE#UPDATED                               NUMBER
```

Multi-value association – *HARDWARE* – which generates the table PLSCUSTOMER#HARDWARE

```
SQL> desc PLSCUSTOMER#HARDWARE
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ID                                        NOT NULL VARCHAR2(10)
 VMC_HARDWARE                              NOT NULL NUMBER
 HARDWARE                                           VARCHAR2(15)
 OS                                                 VARCHAR2(15)
 SYSTEMTYPE                                         VARCHAR2(24)
 NUMUSERS                                           VARCHAR2(6)
 LASTMAINT                                          VARCHAR2(11)
 VMC_HARDWARE#UPDATED                               NUMBER
```

Sub-value association – *UPDATED* – within *HARDWARE* which generates the table PLSCUSTOMER#HARDWARE#UPDATED

```
SQL> desc PLSCUSTOMER#HARDWARE#UPDATED
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ID                                        NOT NULL VARCHAR2(10)
 VMC_HARDWARE                              NOT NULL NUMBER
 VMC_HARDWARE#UPDATED                      NOT NULL NUMBER
 LASTMAINT                                          VARCHAR2(11)
```

Adding some simple data to the table using the jBASE Editor:

```
NEW *File PLSCUSTOMER , Record '00000001'
Command-> fi
001 DONNA
002 JOHNSON
003 1 SUN AVENUE
004
005 SPRINGFIELD
006 OR
007 12345
008 (503) 246-2317
009 (555) 555-1237
010 DONNAJ@forestrbad.com
011 HP]INTEL PII
012 SOLARIS]OSF1
013 jBASE]ROS
014 722]126
015 10000\12345]13001
016 9999
----------------------------------------------------------- End Of Record
```

The data in Oracle shows the data in these tables (shown in a *sqlplus* session):

```
SQL> select * from PLSCUSTOMER;

ID         FIRSTNAME                LASTNAME             ADDR1                ADDR2                CITY        STA ZIP          HOM
ETEL       WORKTEL          EMAIL                    HARDWARE        OS              SYSTEMTYPE              NUMUSE LASTMAINT
BALANCE VMC_HARDWARE VMC_HARDWARE#UPDATED
---------- ------------------------ -------------------- -------------------- -------------------- ----------- --- ------------ ---
------------- ---------------- ------------------------ --------------- --------------- ----------------------- ------ -----------
 ---------- ------------ --------------------
00000001   DONNA                    JOHNSON              1 SUN AVENUE                              SPRINGFIELD OR  12345        (50
3) 246-2317   (555) 555-1237   DONNAJ@forestrbad.com     HP              SOLARIS         jBASE                    722    10000
  99.99         2                    2

SQL> select * from PLSCUSTOMER#HARDWARE;

ID         VMC_HARDWARE HARDWARE        OS              SYSTEMTYPE              NUMUSE LASTMAINT   VMC_HARDWARE#UPDATED
---------- ------------ --------------- --------------- ----------------------- ------ ----------- --------------------
00000001              2 INTEL PII       OSF1            ROS                     126    13001                          1

SQL> select * from PLSCUSTOMER#HARDWARE#UPDATED;

ID         VMC_HARDWARE VMC_HARDWARE#UPDATED LASTMAINT
---------- ------------ -------------------- -----------
00000001              1                    2 12345

SQL>
```

APPENDIX B - DB2EXP DRIVER

Introduction

The DB2EXP jDK driver bridges the gap between jBASE and DB2 with little or no intervention of the jBASE application developer, so that it is transparent for jBASE applications to access DB2 database. Moving the data from the traditional 'hash file' environment to the RDBMS (e.g. DB2) environment brings with it one intrinsic point of which a user should be aware. The 'multi-value' environment is different to the 'RDBMS' environment in the sense that the underlying management of data is different. From a high-level user perspective, they appear (and in fact are) both the same - "the application retrieves data from the database". The method by which the database drivers deliver and manipulate the data 'under the covers' are however very different. Put simply – performing a 'COUNT' operation in one manufacturers database will yield the same results as performing a 'COUNT' on the same data in a different manufactures database, but the method by which the count was generated will differ.

DB2 Requirements

Server Requirements

You must run the EXCEPTIONS.sql script if you plan to use the EXCEPTIONS table (discussed later).

Client Requirements

The DB2 (database) server and jBASE server can reside on the same machine or on separate machines. If the DB2 (database) server and jBASE servers reside on separate machines, it will be necessary to install the DB2 Client software on the jBASE server.

DB2 Configuration

DB2 Database Objects

Create the following objects on the target database:

Tablespaces

- It is recommended that you identify the tablespaces available for the users that will be connecting to the DB2 database. You may need to created additional tablespaces with larger page sizes than the default installation.

Exception Table (EXCEPTIONS.sql)

- Should be run using the db2 shell command connecting as the user you plan to use.

Driver Installation

The major components of the DB2EXP driver include:

libDB2EXP.so and libDB2EXP.so.el (Windows version libDB2EXP.dll and libDB2EXP.def)	The actual driver files. These files need to reside in the jBASE 'lib' directory ($JBCRELEASEDIR/lib) or an appropriate directory in the $JBCOBJECTLIST path.
jEDIdrivers. (On non-Windows platforms this would have a leading "." if placed in the user's $HOME)	Configuration file for holding database and parameter information.
jDK tool (comprising of executables and libraries).	These programs should reside in the jBASE 'bin' directory ($JBCRELEASEDIR/bin) and jBASE 'lib' directory ($JBCRELEASEDIR/lib). Equivalent bin (included in $PATH) and lib (included in $JBCOBJECTLIST) will suffice.

Once the driver has been installed, modify the user's environment to allow the driver to operate. Set the following environment variables:

PATH	Set to include the '$JBCRELEASEDIR/bin' directory
JEDI_SOB_NOCLOSE	Set to the value '1' for correct driver operation

As a generic test, if the user can begin an 'db2' session, then the DB2EXP driver should operate correctly.

Driver Configuration

Before the driver can operate, you must create the configuration file 'jEDIdrivers.ini'. This file physically holds the configuration parameters, which the driver reads.
Create the 'jEDIdrivers.ini' file in one of two places:

- /etc
- In the user's home directory ($HOME)

It is recommended that you create the file in the "/etc" directory. This allows multiple users access to the same configuration. It is possible to have both whereby the $HOME version will compliment and override settings from the /etc version.

jEDIdrivers Configuration

The configuration for the DB2EXP driver is controlled by a [DB2EXP] section heading in jEDIdrivers.ini. The minimum requirements for connecting to the target DB2 database are:

default = <database_connection_string>

The *database_connection_string* could be something like:

DSN=TOOLSDB;UID=peterf;DATABASE=TOOLSDB

Note, *DATABASE* can be omitted and the default database will be used.

The **default** parameter is the database connection identifier. You can have as many of these as you like but it is recommended you have a **default** so that you do not need to specify "…CONNECT=…" in the CREATE-FILE command. Additional headers could be:

testdb = DSN=TOOLSDB;UID=peterf;DATABASE=TESTDB

You can specify the password in the connection string or on its own (mainly if you need it encrypted). In this case an additional header is required which would be DB2EXP_*database_identifier*.

e.g.

[DB2EXP_default]
passwd = mypassword

[DB2EXP_testdb]
PasswdsEncrypted = 1
passwd = Wlm9Avx8+AI=

If the PasswdsEncrypted=1 (either locally under [DB2EXP] or [DB2EXP_*database_identifier*] or globally under [General]) the <password> *must* be encrypted. Refer to the first part of this manual for information on encrypting password.

The following settings may also be required if you need to tailor the rdbms column types for the appropriate csv types (D, T, TS, C, A, N0, N*n*, TXT):
date, time, timestamp, char, varchar, number, decimal, text
e.g. text = BLOB

These types are pre-configured with default values.

Using the Driver

This section lists the commands that are specific to the DB2EXP Driver or have extensions for the DB2EXP Driver.

Definitions

Before you create a DB2EXP type file, you must have a *CSV* definition to map each record from dynamic array to relational. For details on this, refer to the first part of this manual.

CREATE-FILE

Syntax

CREATE-FILE *filename* TYPE=DB2EXP {TABLE=*tablename*} {CSV=*csvdefinition*} {EXISTING=YES} {CONNECT=*database_identifier*}

tablename defaults to filename
csvdefinition defaults to tablename (this definition is read from the directory specified by the **CSVdir** parameter in the jEDIdrivers.ini file).
Any '.' characters in the filename or columns will be converted to a '_' character in the database
database_indentifier defaults to "default" (refer to the section *jEDIdrivers Configuration*).

If *EXISTING* is used then the table is assumed to be an existing table and therefore will not be created (or dropped) during CREATE-FILE/DELETE-FILE.

Example:

```
CREATE-FILE DB2CUSTOMER TYPE=DB2EXP CSV=JCUSTOMER
```

(Executed from a telnet session)

```
mercury-~: CREATE-FILE DB2CUSTOMER TYPE=DB2EXP CSV=JCUSTOMER
[ 417 ] File DB2CUSTOMER]D created , type = DB2EXP
[ 417 ] File DB2CUSTOMER created , type = DB2EXP
mercury-~:
```

NOTE:

That although the dictionary portion of the above file – DB2CUSTOMER]D – is displayed as type = DB2EXP it is in fact a regular j4 hashed file. You could if you wanted to create them separately thus:

```
CREATE-FILE DICT DB2CUSTOMER 1
CREATE-FILE DATA DB2CUSTOMER TYPE=DB2EXP CSV=JCUSTOMER
```

This create-file command does three things:

- Creates a dictionary for the file (unless DATA has been specified)
- creates the necessary table(s) on DB2.
- Writes out a *stub* file – DB2CUSTOMER – to the current working directory which will look like this:

  JBC__SOB JediInitDB2EXP DB2CUSTOMER

In the above example our CSV looked like this:

Column	Attr	Width	Type	Association	Controlling
ID,	0,	10,	AN		
FIRSTNAME,	1,	24,	AN		
LASTNAME,	2,	20,	AN		
ADDR1,	3,	21,	AN		
ADDR2,	4,	20,	AN		
CITY,	5,	11,	AN		
STATE,	6,	3,	AN		
ZIP,	7,	12,	AN		
HOMETEL,	8,	16,	AN		
WORKTEL,	9,	16,	AN		
EMAIL,	10,	25,	AN		
HARDWARE,	11,	15,	AN,	HARDWARE,	1
OS,	12,	15,	AN,	HARDWARE	
SYSTEMTYPE,	13,	24,	AN,	HARDWARE	
NUMUSERS,	14,	6,	AN,	HARDWARE	
LASTMAINT,	15,	11,	DT,	HARDWARE|UPDATED,	1
BALANCE,	16,	12,	N2		

This definition implies three tables:

Primary table (DB2CUSTOMER) which holds all the fields with no *Association*

```
db2 => describe table DB2CUSTOMER

Column                         Type      Type
name                           schema    name               Length   Scale Nulls
------------------------------ --------- ------------------ -------- ----- ------
ID                             SYSIBM    VARCHAR                  10     0 No
FIRSTNAME                      SYSIBM    VARCHAR                  24     0 Yes
LASTNAME                       SYSIBM    VARCHAR                  20     0 Yes
ADDR1                          SYSIBM    VARCHAR                  21     0 Yes
ADDR2                          SYSIBM    VARCHAR                  20     0 Yes
CITY                           SYSIBM    VARCHAR                  11     0 Yes
STATE                          SYSIBM    VARCHAR                   3     0 Yes
ZIP                            SYSIBM    VARCHAR                  12     0 Yes
HOMETEL                        SYSIBM    VARCHAR                  16     0 Yes
WORKTEL                        SYSIBM    VARCHAR                  16     0 Yes
EMAIL                          SYSIBM    VARCHAR                  25     0 Yes
BALANCE                        SYSIBM    DECIMAL                  12     2 Yes
VMC#HARDWARE                   SYSIBM    INTEGER                   4     0 Yes

  13 record(s) selected.
```

Multi-value association – *HARDWARE* – which generates the table

DB2CUSTOMER#HARDWARE

```
Column                         Type      Type
name                           schema    name               Length   Scale Nulls
------------------------------ --------- ------------------ -------- ----- ------
ID                             SYSIBM    VARCHAR                  10     0 No
VMC#HARDWARE                   SYSIBM    INTEGER                   4     0 No
HARDWARE                       SYSIBM    VARCHAR                  15     0 Yes
OS                             SYSIBM    VARCHAR                  15     0 Yes
NUMUSERS                       SYSIBM    VARCHAR                   6     0 Yes
VMC#HARDWARE#SYS               SYSIBM    INTEGER                   4     0 Yes
VMC#HARDWARE#UPDATED           SYSIBM    INTEGER                   4     0 Yes

  7 record(s) selected.

db2 => █
```

Sub-value association – *UPDATED* – within *HARDWARE* which generates the table

DB2CUSTOMER#HARDWARE#UPDATED

```
db2 => describe table DB2CUSTOMER#HARDWARE#UPDATED

Column                         Type      Type
name                           schema    name               Length   Scale Nulls
------------------------------ --------- ------------------ -------- ----- ------
ID                             SYSIBM    VARCHAR                  10     0 No
VMC#HARDWARE                   SYSIBM    INTEGER                   4     0 No
VMC#HARDWARE#UPDATED           SYSIBM    INTEGER                   4     0 No
LASTMAINT                      SYSIBM    DATE                      4     0 Yes

  4 record(s) selected.

db2 => █
```

Adding some simple data to the table using the jBASE Editor:

```
File DB2CUSTOMER , Record '00001'                                                                    Insert
Command->
0001 DONNA
0002 JOHNSON
0003 1 SUN AVENUE
0004
0005 SPRINGFIELD
0006 OR
0007 12345
0008 (503) 123-4567
0009 (503) 321-9876
0010 DONNA@forestbad.com
0011 HP]INTEL PII
0012 SOLARIS]OSF1
0013 jBASE]ROS
0014 722]126
0015 10000\12345]13001
0016 9999
---------------------------------------------------------- End Of Record ---------------------------------------------
```

The data in DB2 shows the data in these tables (shown in a *db2* session):

```
db2 =>
db2 => select * from DB2CUSTOMER

ID         FIRSTNAME                LASTNAME             ADDR1                ADDR2                CITY        STATE ZIP          H
OMETEL          WORKTEL          EMAIL                     BALANCE        VMC#HARDWARE
---------- ------------------------ -------------------- -------------------- -------------------- ----------- ----- ------------ -
---------------- ---------------- ------------------------- -------------- ------------
00001      DONNA                    JOHNSON              1 SUN AVENUE         -                    SPRINGFIELD OR    12345        (
503) 123-4567   (503) 321-9876   DONNA@forestbad.com                99.99            2

  1 record(s) selected.

db2 => select * from DB2CUSTOMER#HARDWARE

ID         VMC#HARDWARE HARDWARE        OS              NUMUSERS VMC#HARDWARE#SYS VMC#HARDWARE#UPDATED
---------- ------------ --------------- --------------- -------- ---------------- --------------------
00001                 1 HP              SOLARIS         722                     1                    2
00001                 2 INTEL PII       OSF1            126                     1                    1

  2 record(s) selected.

db2 => select * from DB2CUSTOMER#HARDWARE#updated

ID         VMC#HARDWARE VMC#HARDWARE#UPDATED LASTMAINT
---------- ------------ -------------------- ----------
00001                 1                    1 05/18/1995
00001                 1                    2 10/18/2001
00001                 2                    1 08/05/2003

  3 record(s) selected.

db2 =>
```

APPENDIX C - OLESQL DRIVER

Introduction

The OLESQL jDK driver attempts to bridge the gap between jBASE and SQL Server with little or no intervention of the jBASE application developer. Moving the data from the traditional 'hash file' environment to the RDBMS (e.g. SQL Server) environment brings with it one intrinsic point of which a user should be aware. The 'multi-value' environment is different to the 'RDBMS' environment in the sense that the underlying management of data is different. From a high-level user perspective, they appear (and in fact are) both the same - "the application retrieves data from the database". The method by which the database drivers deliver and manipulate the data 'under the covers' are however very different. Put simply – performing a 'COUNT' operation in one manufacturers database will yield the same results as performing a 'COUNT' on the same data in a different manufactures database, but the method by which the count was generated will differ.

SQL Server Requirements

Server Requirements

- SQL Server version must be release 7 or higher.

Client Requirements

- You must install the OLESQL driver on the server that has the jBASE installation.
- The SQL Server (database) server can also be on the same machine or on a separate machine on the same network.
- If the SQL Server (database) server is installed on a separate machine to the jBASE server, it will be necessary to install the SQL Server Client software on the jBASE server.

SQL Server Configuration

It is recommended that you use a case sensitive database to enable the different treatment of record keys like 'A' and 'a'.

SQL Server Database Objects

Create the following objects on the target database:

User

Use any user name and password. The driver from the configuration file reads these credentials.

If desired use Windows authentication.

DatabaseObjects (CreateExceptions.sql)

The OLESQL driver may require an EXCEPTIONS table and accompanying stored procedures installed on the database.

You can find the 'CreateExceptions.sql' file in the OLESQL 'zip/tar' file, which can be run from the Query Analyzer or by using the "osql" command.

Installation

The major components of the OLESQL driver include:

libOLESQL.dll and libOLESQL.def	The actual driver files. These files need to reside in the jBASE 'lib' directory (%JBCRELEASEDIR%\lib) or an appropriate directory in the %JBCOBJECTLIST% path.
jEDIdrivers.ini	Configuration file for holding database and parameter information.
jBuildOLE.exe	Program used to generate the stored procedures on SQL Server
Templates	A directory holding skeleton stored procedures used by jBuildOLE
jDK tool set (comprising of executables and libraries).	These programs should reside in the jBASE 'bin' directory (%JBCRELEASEDIR%\bin) and jBASE 'lib' directory (%JBCRELEASEDIR%\lib). Equivalent bin (included in %PATH%) and lib (included in %JBCOBJECTLIST%) will suffice.

Once the driver has been installed, modify the users' environment to allow the driver to operate. Set the following environment variables:

%PATH%	Set to include the '%*SQL_SERVER_HOME*%\bin' directory
%JEDI_SOB_NOCLOSE%	Set to the value '1' for correct driver operation

As a generic test, if the user can begin an 'osql' session, then the OLESQL driver should operate correctly.

Configuration

Before the driver can operate, you must create the configuration file 'jEDIdrivers.ini'. This file physically holds the configuration parameters, which the driver reads.
Create the 'jEDIdrivers.ini' file in one of two places:

%SYSTEMROOT%

In the users home directory (%HOME%)

jEDIdrivers Configuration

The configuration for the OLESQL driver is controlled by a [OLESQL] section heading in jEDIdrivers.ini. The minimum requirements for connecting to the target SQL Server database is:

user = <SQL_Server_user>
passwd = <password>
server = <host_address>   {name or IP address of server if not the local host}
database = <SQL Server database>   {If SQL Server database is exported then this is not required}

If the PasswdsEncrypted=1 (either locally under [OLESQL] or globally under [General]) the <password> must be encrypted. Refer to the first part of this manual for information on encrypting password.

It is recommended that you create the file in the %SYSTEMROOT% directory. This allows multiple users access to the same configuration.

Using the Driver

This section lists the commands that are specific to the SQL Server Driver or have extensions for the SQL Server Driver.

Definitions

Before you create an OLESQL type file, you must have a *CSV* definition to map each record from dynamic array to relational. For details on this, refer to the first part of this manual.

CREATE-FILE

Syntax

CREATE-FILE *filename* TYPE=OLESQL {TABLE=*tablename*} {CSV=*csvdefinition*} {EXISTING=YES} {WRITEOPTS=*options*} {NOWANCHAR=YES}

tablename defaults to filename

csvdefinition defaults to tablename (this definition is read from the directory specified by the **CSVdir** parameter in the jEDIdrivers.ini file).

Any '.' characters in the filename or columns will be converted to an '_' character in the database

If *EXISTING* is used then the table is assumed to be an existing table and therefore will not be created (or dropped) during CREATE-FILE/DELETE-FILE.

options can be any combination of I (insert), U (update) or D (delete) to restrict updates to the table. This is normally used with the *EXISTING* option when interfacing to a table which is not "owned" by the jBASE application.

The **NOWANCHAR=YES** option allows the creation of old (pre-wide character support) style files for backwards compatibility with drivers older than version 4.0.10.

Example:

```
CREATE-FILE OLECUSTOMER TYPE=OLESQL CSV=JCUSTOMER
```

(Executed from a telnet session)

```
jsh peterf ~ -->CREATE-FILE OLECUSTOMER TYPE=OLESQL CSV=JCUSTOMER
[ 417 ] File OLECUSTOMER]D created , type = OLESQL
[ 417 ] File OLECUSTOMER created , type = OLESQL
jsh peterf ~ -->
```

NOTE:

That although the dictionary portion of the above file – OLECUSTOMER]D – is displayed as type = OLESQL it is in fact a regular j4 hashed file. You could if you wanted to create them separately thus:

```
CREATE-FILE DICT OLECUSTOMER 1
CREATE-FILE DATA OLECUSTOMER TYPE=OLESQL CSV=JCUSTOMER
```

This create-file command does three things:

Creates a dictionary for the file (unless DATA has been specified)

Generates a script to run against SQL Server by executing the ***jBuildOLE*** command against the csv file: JCUSTOMER.csv

Runs the script, which creates the necessary table(s) and stored procedures.

Writes out a *stub* file – OLECUSTOMER – to the current working directory which will look like this:

JBC__SOB JediInitOLESQL OLECUSTOMER csv=JCUSTOMER {WANCHAR}

In the above example our CSV looked like this:

Column	Attr	Width	Type	Association	Controlling
ID,	0,	10,	AN		
FIRSTNAME,	1,	24,	WN		
LASTNAME,	2,	20,	WN		
ADDR1,	3,	21,	AN		
ADDR2,	4,	20,	AN		
CITY,	5,	11,	AN		
STATE,	6,	3,	AN		
ZIP,	7,	12,	AN		
HOMETEL,	8,	16,	AN		
WORKTEL,	9,	16,	AN		
EMAIL,	10,	25,	AN		
HARDWARE,	11,	15,	AN,	HARDWARE,	1
OS,	12,	15,	AN,	HARDWARE	
SYSTEMTYPE,	13,	24,	AN,	HARDWARE	

```
NUMUSERS,      14,   6,     AN,   HARDWARE
LASTMAINT,     15,   11,    DT,   HARDWARE|UPDATED,   1
BALANCE,       16,   12,    N2
```

This definition implies three tables

Primary table (OLECUSTOMER which holds all the fields with no *Association*

Multi-value association – *HARDWARE* – which generates the table OLECUSTOMER#HARDWARE

Sub-value association – *UPDATED* – within *HARDWARE* which generates the table OLECUSTOMER#HARDWARE#UPDATED

Adding some simple data to the table using the jBASE Editor:

```
jsh peterf ~ -->ED OLECUSTOMER 0000001
0000001
TOP
.P
TOP
001 DONNA
002 JOHNSON
003 1 SUN AVENUE
004
005 SPRINGFIELD
006 OR
007 12345
008 (503) 246-2317
009 (555) 555-1237
010 DONNAJ@forestrbad.com
011 HP]INTEL PII
012 SOLARIS]OSF1
013 jBASE]ROS
014 722]126
015 10000\12345]13001
016 9999
BOTTOM
.FI
Record '0000001' written to file 'OLECUSTOMER'
jsh peterf ~ -->
```

The data in SQL Server shows the data in these tables (shown in an *osql* session):

```
1> SELECT * FROM OLECUSTOMER
2> GO
 id         FIRSTNAME               LASTNAME             ADDR1                ADDR2                CITY        STATE ZIP

       HOMETEL          WORKTEL          EMAIL                     BALANCE        UMC#HARDWARE
 ---------- ----------------------- -------------------- --------------------- -------------------- ----------- ----- ----------

       ---------------- ---------------- ------------------------- -------------- ------------
 0000001    DONNA                   JOHNSON              1 SUN AVENUE         NULL                 SPRINGFIELD OR    12345

       (503) 246-2317   (555) 555-1237   DONNAJ@forestrbad.com             99.99            2

(1 row affected)
1> SELECT * FROM OLECUSTOMER#HARDWARE
2> GO
 id         UMC#HARDWARE HARDWARE        OS              SYSTEMTYPE              NUMUSERS UMC#HARDWARE#UPDATED
 ---------- ------------ --------------- --------------- ----------------------- -------- --------------------
 0000001               1 HP              SOLARIS         jBASE                   722                         2
 0000001               2 INTEL PII       OSF1            ROS                     126                         1

(2 rows affected)
1> SELECT * FROM OLECUSTOMER#HARDWARE#UPDATED
2> GO
 id         UMC#HARDWARE UMC#HARDWARE#UPDATED LASTMAINT
 ---------- ------------ -------------------- -----------------------
 0000001    1                               1 1995-05-18 00:00:00.000
 0000001    1                               2 2001-10-18 00:00:00.000
 0000001    2                               1 2003-08-05 00:00:00.000

(3 rows affected)
1>
```

APPENDIX C - FREQUENTLY ASKED QUESTIONS

How does jBASE find the driver?

When installing the driver, it deploys two files to a 'library' directory. This directory path must be included in the environment variable 'JBCOBJECTLIST' unless the files are copied into the $JBCRELEASEDIR/lib directory. These two files are the driver, which has an extension of '.so' and a text file (with the same name as the driver file with a '.el' extension). The text file lists the 'entry' points into the driver that jBASE can use. If jBASE is unable to locate the text file and driver, it displays a generic jBASE error:

```
** Error [ JEDI_FILEOP_ILLEGAL_CMD ] **
Illegal file operation command passed to jedi
```

If this occurs then check the parameters supplied on the command line to ensure they are correct and then use a command like 'jshow -v JediInitPLSORA' and check that the driver is listed in the returning output. If not, then the 'JBCOBJECTLIST' has not been set up correctly or the driver is not in the $JBCRELEASEID/lib.

NOTE:

Due to incorrect file transfer methods, where it replaces 'Carriage Return' characters with 'Carriage Return and Line Feed' characters corruption to the '.el' or '.def' file can occur. Another reason why the driver may fail to load is due to dependencies required by the driver, which are not found in the library or bin paths of the current environment.

How does jBASE use the driver?

The jBASE External Database Interface (jEDI) is the technology that jBASE uses to manipulate data in external databases (e.g. Oracle). This technology implements a published API interface. Each driver (including the PLSORA driver) must provide the following functions:
Common jEDI Interface Routines:

Initialise	IOCTL
Open	Clear File

Close	Delete File
Select	Synchronisation
Select End	Transaction Begin
Read Next	Transaction Rollback
Read	Transaction Commit
Write	
Delete	
Lock	

Some of these routines (typically SELECT, READ, READNEXT, WRITE and IOCTL) are discussed later in the document. It is important to note the difference in paradigms between jBASE and Oracle methodology. For example – Unlike in Oracle, jBASE has no concept of 'Update or Insert' there is simply a 'WRITE' routine.

What happens if I try to write invalid data?

The CSV definition used when creating the file – and table(s) – determines what data types are valid and which fields are repeating. Unlike jBASE, it can only store values that are consistent with these data types. The PLSORA driver has built-in logic for handling invalid records. Whether using this or not is determined by the **TrapErrs** parameter in the .jEDIdrivers.ini. Setting this variable to '1' ensures that all writes will succeed regardless of whether the record is consistent with the CSV structure or not. The handling of this is via an ***EXCEPTIONS*** table (created as part of the installation), which holds the following fields:

filename
item-id
update timestamp
program which performed the write
dynamic array record image

When a jBASE application tries to read back this record it first attempts to read from the intended table. If not found it looks in the ***EXCEPTIONS*** table and the application does not perceive any difference in logic flow.

How does a SELECT work if there are *EXCEPTIONS*?

A *SELECT* from jBASE indicates a *SELECT* from the primary table. If **TrapErrs** is set to '1' then it includes a ***UNION ALL*** SELECT on the EXCEPTIONS table based on the current filename.

How does Exception handling affect performance?

The problem with relying on **TrapErrs** to catch your mistakes is that there is extra work needed by the driver to ensure data integrity. The following points highlight this:

For every read where the record is not found on the primary table, it performs an additional SELECT on the ***EXCEPTIONS*** table (the same goes for **deleting** records).

For writes, the driver tries to detect the use of invalid data, and then calls the stored procedure to handle and update the table(s). If any update fails the driver has to then update the ***EXCEPTIONS*** table.

If performing a write on a table with no repeating groups it needs to check if the record was previously in the ***EXCEPTIONS*** table and if so delete it.

Any SELECT has to include the ***EXCEPTIONS*** table to ensure retrieval of all records.

What if I don't want the *EXCEPTIONS* table?

Setting **TrapErrs** to zero means that an invalid write from jBASE will cause an error. The offending program can be coded with an ***ON ERROR*** clause and handle the error programmatically. If not coded the application will see a write error message with the options:
(I)gnore, (R)etry, (Q)uit

I have a parameter file where records are of different formats. How can I store that ?

If you use an empty CSV file as the definition for your table, it creates the records with a generic record_id and record image (i.e. the dynamic array); referred to as a 'LOB'.

How can I see what the driver is doing?

There are two tracing methods: screen and logfile. First, you must set a trace level:

Depending on the driver you are using, set the appropriate trace environment variable where <drivername> is set to the appropriate driver (PLSORA or DB2EXP)

```
Export JEDI_<drivername>_TRACE=1..4 or
```

For a basic trace of I/O set this environment variable to '1', for a more detailed trace use '2' , and so on.
Next, you optionally can define:

`Export JEDI_<drivername>_LOG=1 and export JEDI_<drivername>_LOGFILE=`*`logfilename`*` to the name of the file you want to store the trace information.`

You can also set:

`Export JEDI_<drivername>_DISPLAY=1 to display the log to the screen at the same time.`

NOTE: if you do not set JEDI_<drivername>_LOG then JEDI_<drivername>_DISPLAY is assume

What stored procedures does Oracle use?

When creating a file, it creates the following stored procedures:

Tablename_PKG.READ *

Tablename_PKG.WRITE *

Tablename_PKG.SEL *

Tablename_PKG.DELETE *

Tablename_PKG.DELREC*

Tablename_PKG.LOCKITEM*

* it creates these stored procedures for *every* table (i.e. including repeating group associated tables).

Additionally if using ***EXCEPTIONS***, the stored procedures **READERROR** (used for reads), **WRITEERROR** (used for writes).

What stored procedures does SQL server use?

When creating a file, it creates the following stored procedures:

Tablename_OPEN *

Tablename_READ *

Tablename_WRITEPREP

Tablename_WRITE *

Tablename_DELETE*

Tablename_CLEAR

Tablename_DROP

Tablename_OPENCURS

Tablename_FETCHCURS

Tablename_CLOSECURS

* it creates these stored procedures for *every* table (i.e. including repeating group associated tables).

Additionally if using ***EXCEPTIONS***, the stored procedures **READ_EXCEPTION** (used for reads), **WRITE_EXCEPTION** and **GET_EXCEPTION** (used for writes) and **DELETE_EXCEPTION** (used whenever a deletion is required) may be required.

What happens when the driver reads?

It prepares and executes an SQL SELECT for the primary table. If the result returns multi-value counts > 0 (or > 1 if FullyExpanded < 2) then additional SELECT statements are prepared and executed for the associated (multi-value/sub-value) tables. The driver then fetches from each 'cursor' and assembles the dynamic array. Reads are also involved in jBASE SELECTs. The prepare step is only done the first time a statement is executed (while the file handle remains open).
Imagine the following pseudo code program example:
NOTE: The example below is only one method of retrieving and reading data. There are other methods of using jBASE to extract and read the data.

SELECT data	Create a select list of ID's matching any WHERE/WITH criteria
READNEXT key	Read the first\next ID from the select list
READ data	Read the actual data record
READNEXT key	Read the next key in the select list
CLOSE	Close the select list and connection to the database

When creating a SELECT list, it opens a cursor and maintains it on the Oracle database.
The three common jEDI interface routines used in the driver for reading are SELECT, READNEXT and READ.

SELECT

When jBASE issues a SELECT command to the driver, the driver needs to read a list of record keys from the database and store them in a ‘select list’ (this is a jBASE term for a list of ID’s).
In the same way that an RDBMS SQL statement can contain a WHERE clause to limit the number of records returned to only those that match a given criteria, a jBASE jQL statement (SELECT) can also have a WITH clause that performs the same restrictions.

jQL Statement - SELECT <tablename>
SQL Statement - SELECT ID FROM <tablename>
jQL Statement - SELECT <tablename> WITH ID=’BOB’
SQL Statement - SELECT ID FROM <tablename> WHERE ID=’BOB’

The above examples, which show the used jQL/SQL clause can have a large impact on the way the ‘READ’ operation performs.
The logic for the SELECT routine is as follows:

Open a cursor on the primary key (if the SELECT from jBASE is a jQL select with criteria on the record itself then the cursor must also include the columns for the entire record).
Fetch each ‘row’ from the cursor.
Pass the result to the jQL parser to determine if the record matches the query.

READNEXT

After generating the select list, jBASE calls the READNEXT routine to get the next ID from the select list. The SELECT or OPEN routine previously called will have an open select list (‘cursor’ on the Oracle database) and the routine performs a ‘fetch’ action to read the next ID in the cursor.

READ

After a successful ‘READNEXT’ operation, it passes the current returned ID to the READ routine to read the actual data record, which might not match any criteria specified in a jQL SELECT WITH clause. However, jBASE still needs to read the record from Oracle to check whether it meets with the WITH clause criteria. It is easy to see how inefficient this method could potentially be if the criteria used only matches a few records from a large table.
The current design of the driver means that to perform some SQL statements through the driver can take considerably longer than issuing them directly through Oracle. To clarify, the best example is the jQL

statement COUNT <tablename> and the corresponding SQL statement SELECT COUNT(ID) FROM <tablename>. When issuing this command in a Oracle client (e.g. sqlplus) the Oracle SQL parser and engine will not individually count each record instead, a count using the index is "made". However, in jBASE to perform the same action, the jBASE driver performs a SELECT to open the cursor and then performs individual READNEXT's on each record ID to count the records.

This would mean that the SQL COUNT would complete a lot quicker than a jQL COUNT because they are performing different actions. For a more accurate comparison, change the SQL statements to force an Oracle to count in the same manor as jBASE. If the SQL statement was changed in both cases to use a WHERE or WITH clause on a non-indexed column, then both jBASE and Oracle would have to perform a read on each record to check if the record matched the WHERE or WITH criteria.

The example for this question used a SELECT, READNEXT and READ cycle. This is not the only way that a read may occur. For example, it may not be necessary to use SELECT and READNEXT. If the ID is already known by the calling program, then the driver may only need to call the OPEN routine (which does not have database access) and then a straight READ given the ID.

A Better way to handle SELECT

One way to speed up any jBASE SELECT is to issue an 'IOCTL' call to the driver prior to executing the SELECT (this can be an 'external' or 'jQL' select or a jBASIC 'internal' select).

e.g.

```
INCLUDE JBC.h
OPEN 'OLECUSTOMER' TO F.PLSORA ELSE STOP 201,'OLECUSTOMER'
mysqlstmt = "SELECT ID FROM OLECUSTOMER WHERE LASTNAME LIKE 'SMITH%'"
rc = IOCTL(F.PLSORA, JIOCTL_COMMAND_SQL_SELECT_PREPARE, mysqlstmt)
SELECT F.PLSORA
LOOP WHILE READNEXT id DO
...
REPEAT
```

The select clause can have any criteria or sort order but must only return the primary key column(s) (in the case of more than one primary key column they must be specified in the order they were specified in the CSV used to create the file/table).

What happens when the driver writes?

There are two scenarios involved when writing records

Writes involving records with associated multi-value/sub-values

Writes with no multi-values (including, obviously, 2-column tables)

For those involving multi-values the first step in the update process is to execute the WRITE stored procedure for the appropriate record key. This fetches the columns in the primary table that hold the number of multi-values in each associated table. This achieves two things:

It tells the driver that the record exists (assuming the fetch returned a result) which means the driver can issue an UPDATE knowing it will succeed; conversely if the fetch returned a null result the driver knows to INSERT.

Use the number of multi-values for each association that already exists to determine whether to use INSERT, UPDATE or DELETE (in the case of the new record having few multi-values than before).

NOTE

The same goes for sub-values within multi-values except that additional fetches are involved to get the sub-value count for each multi-value.

Next, the driver extracts the relevant data from the dynamic array and calls the WRITE stored procedure for the primary table and associated tables. All these calls are inside a transaction such that if any one of them fails the transaction is 'rolled' back and the exception handling takes place (if **TrapErrs** is set to '1').

In the case of a 2-column table (i.e. where an empty CSV was used), it is simply a matter of calling the WRITE stored procedure. It automatically handles the issue of INSERT and UPDATE (i.e. if the INSERT fails it automatically does an UPDATE without the need of a second trip to the database API).

If the write is successful and an INSERT took place then the **EXCEPTIONS** table may have the original record in which case it calls the **WRITEERROR** stored procedure.

What is the locking strategy?

The current locking strategy is that the jBASE jEDI layer handles all locking. This means that either the jBASE jRLA daemon (if activated) performs the locking, or uses the UNIX file locking mechanism.

The jRLA daemon will lock the file at item level (Record/Row). Unlike RDBMS, locking in a jBASE application only occurs when requested (e.g. using a READU command instead if a READ command). This means that if you open the same record twice in a jBASE editor, it marks the second instance as 'Read Only'.

For further information on jBASE locking strategy, refer to the relevant documentation.

When do records get committed

Unless using transaction boundaries (Begin Trans, Commit Trans and Rollback Trans), records are by default committed after each write.

How are transactions implemented

If a jBASIC program issues a TRANSTART, it sets a flag in the driver to initiate an isolated transaction when jBASIC issues a TRANSEND this transaction is then 'committed'.

When are database connections opened and closed

A Database connection is made the first time the jBASE application opens a file *stub*, which points to the relevant driver and uses this connection for all subsequent file I/O. When all those *stub* type files are closed (typically when the program ends) the connection closes.

How is I18N UTF-8 data handled?

From version 4.0.10 and later the driver handles conversion of UTF-8 data to store I18N data in Unicode data types on the database.

This is achieved by using the '**W**' and '**WN**' Types in the CSV file instead of '**A**', '**C**' and '**AN**'. These must be manually changed if the CSV file is automatically generated.

These types are stored as **nchar** and **nvarchar** on the database, and data conversions are applied to the fields during reads and writes to convert between UTF-8 multibyte encoding in jBASE and Unicode (UCS-2) wide-character encoding in the database. Thus the database can be queried and displayed using regular query tools.

NOTE:

Files created with the new driver have significant changes to the database schema to tell it which fields require conversion. Therefore a new **WANCHAR** (wide and nchar) tag is added to the jBASE stub file to indicate the new format.

Old files created with previous versions of the driver will not have this stub entry, and are read and written to without any conversions taking place.

What does the jstat command show me?

For OLESQL files, the jBASE 'jstat' command shows the driver type and version, the table name and whether the file supports 'W'ide characters (see above). For example, '**jstat OLECUSTOMER**' gives:

```
  Driver type:OLESQL-4.0.10
   Table name:OLECUSTOMER
NCHAR support:YES
```

What do the following errors mean?

Procedure '<tablename>_OPEN' expects parameter '@AttrTypes', which was not supplied.

This error is usually shown in the following sequence (with JEDI_OLESQL_TRACE=2) when trying to access a file e.g. with the command **LIST <filename>**:

```
JEDI_OLESQL_OPEN: {? = call <filename>_OPEN(?, ?, ?)}
Procedure '<filename>_OPEN' expects parameter '@AttrTypes', which was
not supplied.
Failed to execute command (OPEN ).
Error in OPEN
```

This usually means that you are running an old version of the OLESQL driver against a database that was created with the new version. The new driver adds an additional parameter to the _OPEN stored procedure by default, and the old driver does not supply this in the parameter list when it executes the stored procedure.

This is shown by the fact that the _OPEN call only shows three parameters (as question marks) in the trace output.

Failed to create file; Incorrect syntax near '<'

This error is usually shown in the following sequence (with JEDI_OLESQL_TRACE=2) when trying to create a file e.g. with rhe command

CREATE-FILE <filename> TYPE=OLESQL CSV=<csvfilename>:

```
JEDI_OLESQL_CREATE: TableName <filename>
EXECUTE: jBuildOLE <filename> <csvfilename>.csv -O -P
EXECUTE: cmd.exe /C osql -n -U <username> -P <password> -d <databasenam
e> -S <servername> -i c:\jEDI1\temp/OLEDB_<filename>.sql
JEDI_OLESQL_CREATE: Create 'cmd.exe /C osql -n -U <username> -P <passwo
rd> -d <databasename> -S <servername> -i c:\jEDI1\temp/OLEDB_<filename>
.sql' failed for table <filename>, DSN NAVDSN, error Msg 170, Level 15,
 State 1, Server <servername>, Procedure N■Line 5: Incorrect syntax nea
r '<'. <servername>, Procedure <filename>_OPEN, Line 5

** Error [ JEDI_FILEOP_FAIL ] **
```

```
File operation failed, rc -1, for file <filename>
```

The reason for this error is because an old version of the driver utility **jBUILDOLE** was used whilst the template files for the new driver are in place.

Check that you have correctly installed the OLESQL driver package, and configured the environment to use the appropriate version of the driver. Carefully check the **jediDrivers.ini** file.