SQLRDD for xHarbour Builder

A Database Driver for all major SQL database systems

User's guide

© 2004, Marcelo Lombardo
marcelo@xharbour.com.br

Table of Contents

Introduction

SQLRDD was created to access SQL database servers like Oracle, Microsoft SQL Server, IBM DB/2, Postgres, and Sybase, MySQL, from xHarbour xBase language (www.xharbour.com).

You will need to have a fully working installation of one of these supported relational database management system (RDBMS) to make use of SQLRDD.

SQLRDD is more than a RDD. It is a complete suite to interact with SQL databases from within xHarbour compiler.

So, we strongly recommend you to learn about SQL Databases and SQL Statements, as well as the SQL Architecture and topology before using this product. Please, take some time to read the "Getting Started" manual of your preferred database system, and study the SQL commands syntax to define and manipulate data inside the database.

We also recommend that you read this *complete* guide before you start using SQLRDD.

The major features of SQLRDD are:

- Transparent access to SQL databases using xBase commands and functions (DML) like USE, SEEK, ZAP, dbRLock(), etc.
- Creates the database components using xBase Data Definition Language (DDL) like dbCreate(), INDEX ON, etc.
- Easy migration from another RDDs
- Satisfactory performance even with huge tables
- Translates RELATIONS into JOINS or OUTER JOINS inside the database server to have a better use of the SQL Engine features and gain performance in xBase code
- Ability to maintain historic data inside the tables (this is a new feature that has no equivalent feature, in any other RDD engine!). Using historic data you get track of table updates and inserts in the time line, and the actual date determines the visible record. As an example, you can add an employee "next week", and this record will appear to the users on next week. You can raise the price of a product "next month" and this change will only become visible in next month. You can also set the actual date to each workarea. Thus, if you set the actual date to past month or to yesterday, you will see the records as they were last month or

yesterday.

- Code once, run anywhere: The same application runs with all database systems without modifications

- Create royalty-free applications

Requirements

Applications running SQLRDD have the following requirements:

- Standard xHarbour requirements
- Your DBF application requirements
- 16M to 32M RAM per concurrent user under Linux
- 16M to 32M RAM at workstation under Windows
- A dedicated server for the database.
- Database Server hardware requirements (typically a minimal 512M RAM, PIV machine with SCSI hard drive)
- Database Server Software requirements
- ODBC Drivers to the target database (if not using native connection) at workstation or Server (Linux).
- Database client installation (when using Oracle, Ingres or DB2)

Note: the final application performance is dependent of addressing these requirements.

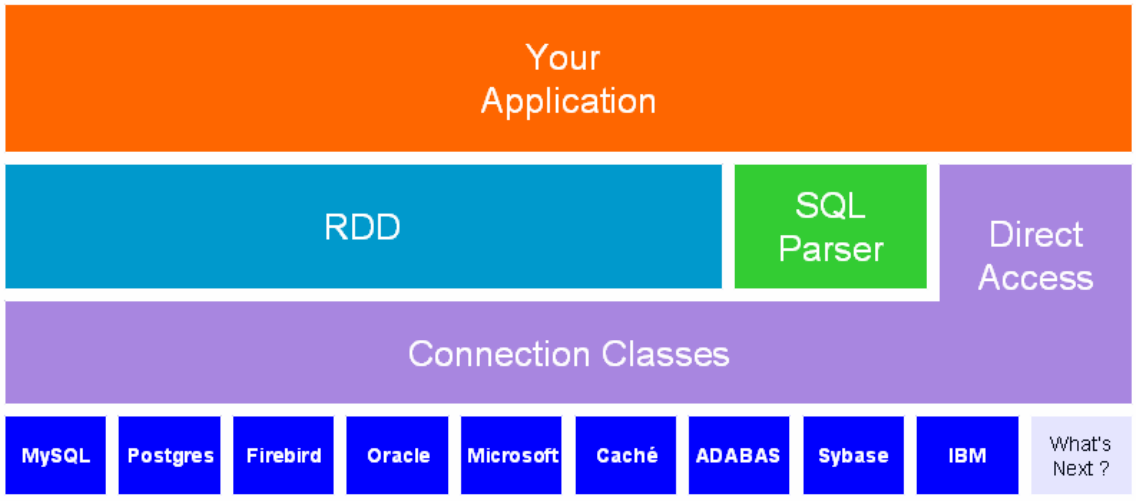
Supported databases

SQLRDD is tested with following databases and connections:

Database	Windows	Linux
Microsoft SQL Server 7, 2000 and 2005	ODBC (supplied by Microsoft) or Native Driver (2005)	Unix ODBC (supplied by third part vendors)
Oracle 8i, 9i, 10g	Native access based on OCI (Oracle Call Interface, supplied by Oracle) or ODBC (supplied by Oracle)	Native access based on OCI (Oracle Call Interface, supplied by Oracle) or Unix ODBC (supplied by third part vendors)
IBM DB2 V 8.x	ODBC (supplied by IBM)	Unix ODBC (supplied by IBM)
Postgres 7.3, 7.4, 8.0	Native Access , ODBC	Native Access, Unix ODBC
MySQL 4.1.x	Native Access, ODBC	Native Access, Unix ODBC
Firebird 1.5 and 2.0	Native Access based in FB Client API or ODBC (supplied by IBPhoenix)	Native Access based in FB Client API or ODBC (supplied by IBPhoenix)
Ingres R3	ODBC (supplied by CA)	Unix ODBC (supplied by CA)
Sybase System XI, XII and Adaptive Server	ODBC (supplied by Sybase)	Unix ODBC (supplied by Sybase)
ADABAS-D	ODBC (Supplied by Software AG)	Unix ODBC (Supplied by Software AG)
Caché	ODBC (Supplied by Intersystems)	Unix ODBC (Supplied by Intersystems)

1. Components

The SQLRDD software layers are:



The main components of SQLRDD:

- **The connection classes**: These are a set of classes used to connect to the database systems. We have different classes for ODBC, Postgres. MySQL and RPC, and more classes will be built in future versions. These classes act as an abstraction layer for the RDD access to the database server, and have the exact same syntax in all of then. With the connection classes, you can also issue SQL Commands direct to the database system and explore the result set, if it exists.

- **The RDD**: The RDD is the low level DML and DDL interface, and it translates the xBase commands into SQL Statements. These statements are issued to the database server through the connection classes.

- **The SQL Parser**: Using the Connection Classes, you can send any command direct to the database system. But unfortunately, the SQL Syntax from the different database vendors is quite different. It means that you have to create a different SQL sentence to every target database, if you want to have a portable application. With the SQL Parser, all you need to get portable with the major database systems is to use the xHarbour SQL Language. The parser “understands” the xHarbour SQL Syntax and generates an intermediate portable code from the SQL phrase. Than you can use the Code Generator to automatically create the specific-system SQL command.

- **The Code Generator**: This component generates the SQL statement from the intermediate portable code created with the SQL Parser. So you code once (using xHarbour SQL) and executes anywhere (with the Code Generator and Connection Classes).

The connection Classes

The Connection Classes are a set of classes used to connect to the different database systems. These classes are used inside the RDD to establish the needed communication between your application and the database.

The programmer **must** use these classes to connect to the database server **before** using any table or SQL function, except the SQL Parser.

We can have several simultaneous connections opened, and select the active connection before using a workarea.

The SQLRDD have some friendly functions to encapsulate all the complexity of the Connection Classes. The most important functions are:

SR_AddConnection(<nType>, <cDSN>) ➔ nHandle

SR_AddConnection creates a new connection to the database system, and set it as active, returning a connection handle.

<nType> may be:

CONNECT_ODBC	(supports Unix ODBC and Windows ODBC32)
CONNECT_RPC	(not available)
CONNECT_MYSQL	(Native support for MySQL >= 4.1)
CONNECT_POSTGRES	(Native support for Postgres >= 7.3)
CONNECT_FIREBIRD	(Native support for Firebird >= 1.5)
CONNECT_ORACLE	(Native support for Oracle 9 and 10)

<cDSN> is a string with connection information such as Host, Database Name, User Name, Password, etc. The string structure is different depending on the **connection type**:

CONNECT_ODBC

“DSN=ODBCdataSourceName;UID=username;PWD=password;DTB=database“

Where:

ODBCdataSourceName : the Data Source Name created in ODBC administrator panel.
username : Database login name
password : Database password

Example: Connect to SQL Server using data source name “Northwind”, username “sa” and a blank password:

```
SR_AddConnection(CONNECT_ODBC, “DSN=Northwind;UID=sa;PWD=“ )
```

CONNECT_MYSQL, CONNECT_POSTGRES and CONNECT_ORACLE

```
“HST=ServerNameOrIP;UID=username;PWD=password;DTB=DataBaseName;PRT=Port“
```

Where:

ServerNameOrIP: The network server name or its IP address.
username : Database login name
password : Database password
DataBaseName: Database name to be used (optional)
Port : TCP Port Number (optional)

Example: Connect to server “localhost”, database “test”, username “root” and password “1234”

```
SR_AddConnection(CONNECT_POSTGRES,
“HST=localhost;UID=root;PWD=1234;DTB=test“ )
```

Note: The “HST” option name can be replaced with the native interface internal name. The internal names are:

OCI -> Oracle Call Interface
PGS -> Postgres
MySQL -> MySQL
FB or FIREBIRD -> Firebird

Examples:

```
SR_AddConnection(CONNECT_POSTGRES,
“PGS=localhost;UID=root;PWD=1234;DTB=test“ )

SR_AddConnection(CONNECT_ORACLE,
“OCI=localhost;UID=system;PWD=manager;DTB=test“ )
```

There is no difference on using “HST” or the internal interface name.

CONNECT_FIREBIRD

Firebird native connection is very similar to the above, but can specify the char set as an option. Example:

```
HST=C:\firebird\DESENV.GDB;uid=SYSDBA;pwd=masterkey;charset=ISO8859_1

Or
```

```
FIREBIRD=C:\firebird\DESENV.GDB;uid=SYSDBA;pwd=masterkey;charset=ISO8859_1
```

SR_DetectDBFromDSN(<cDSN>) or
DetectDBFromDSN(<cDSN>) ➔ ConnectionConstant

Detects the database connection you plan to use based on the internal interface name in DSN. It is planned to be used together with SR_AddConnection() to determine the correct connection constant. Example:

SR_AddConnection(SR_DetectDBFromDSN(cConnString), cConnString)

It checks for OCI=, PGS=, DSN=, MYSQL= , FB= or FIREBIRD= and returns: CONNECT_ORACLE , CONNECT_POSTGRES, CONNECT_ODBC, CONNECT_MYSQL, CONNECT_FIREBIRD respectively.

SR_SetActiveConnection(<nHandle>)

Sets the actual active connection to the next dbUseArea() function, or USE command. This is not needed if the application has only one connection to the database server.

SR_GetActiveConnection() ➔ nHandle

Retrieves the active connection handle

SR_GetConnection(<nHandle>) ➔ oSql

Retrieves the active connection object. You can use the connection object to issue SQL Statements direct to the target database system, using following syntax:

```
oSql:exec( <cCommand>, [<lMsg>], [<lFetch>], [<aArray>], [<cFile>],
[<cAlias>], [<nMaxRecords>], [<lNoRecno>] ) ➔ nRetStatus
```

Where:

<cCommand> is a valid SQL statement to current database.

Note: Please read chapter about SQL Parser / Code Generator if you want to create cross-platform applications
<lMsg> Set it to .F. if you don't want to generate a Run Time error if command fails. Default is .T.

<lFetch> Set it to .T. if you want to fetch the resulting record set. You can choose to fetch row(s) to an array or file. Default is .F.

Note: If you try to fetch the result set of a DDL command or a INSERT, UPDATE or DELETE, database may generate a Run Time Error.

<aArray> An empty array (by reference) to be filled with record set, as follow:

{ { Line1Col1, Line1Col2, … }, { Line2Col1, Line2Col2, … }, … }

<cFile> Is the target filename to output record set. If file is already opened (with alias name like filename) SQLRDD will push result set in this file. File should have exact same structure as result set, or Run Time Error will be raised. If file does not exist, it will be created using default RDD. At the end of operation, file will be opened in exclusive mode and record positioned in Top.

<cAlias> Is an optional alias name to above file.

<nMaxRecords> Is an optional parameter to LIMIT fetch size.

<lNoRecno> If you set to .T., SR_RECNO column will not be fetched to target file or array. Default is .F.

You should declare the needed connection class in your main PRG, choosing one or more from the following options:

```
REQUEST SR_ODBC          /* ODBC ou Unix ODBC Connection */
REQUEST SR_MYSQL         /* My SQL 4.1 direct connection */
REQUEST SR_PGS           /* Postgres >= 7.3 native connection */
REQUEST SR_ORACLE        /* Oracle 8 or higher */
REQUEST SR_FIREBIRD      /* Firebird 1.5 or higher */
```

Please check the Reference Guide for the complete set of functions and Connection Class Methods.

The RDD

SQL is abbreviation of *Structured Query Language*. SQL is a standardized query language for requesting information from a database. The original version called SEQUEL (Structured English Query Language) was designed by an IBM research center in 1974 and 1975. SQL was first introduced as a commercial database system in 1979 by Oracle Corporation.

The RDD is the SQLRDD's component which translates xBase commands into SQL statements, and the SQL result set into workarea data.

This translation might be quiet involved, because DBF tables and SQL tables have different concepts. DBF Tables (and xBase DML in general) have an ISAM-like behavior.

ISAM is abbreviation for *Indexed Sequential Access Method*, a method for managing how a computer accesses records and files stored on a hard disk. While storing data sequentially, ISAM provides direct access to specific records through an index. This combination results in quick data access regardless of whether records are being accessed sequentially or randomly. Thus, indexes are used not only for search data. They define the record order in the same time.

On the other hand, SQL tables are record-set driven. With this technique, you send a query to the database system, and it sends back the records that match your query. Indexes are used for search data and not to order the data. Ordering is a second step internally executed in the database server.

To acquire the best results in translating xBase to SQL, we need to have different approaches to address large and small tables. If we execute a "SELECT * FROM <table>" (it means "give-me all the records from <table>") in a small table, depending on your system, the database server will likely respond instantly. But if we execute the same command on a very large table, you might have to wait a long time for the reply. Further, you might utilize excessive server resources creating a temporary table, with thousands of records and sorting it, but you might end-up using only few of these records in your application.

Thus, SQLRDD have internally two different behaviors in RDD component:

- **Cache workarea**: All the records of the target table are sent to the application and stored in memory. With this method, the xBase workarea runs very fast, because all of the commands are executed in memory, without requesting the database server. But it may be very slow with huge tables, and might utilize excessive

server resources.

- **ISAM Simulator**: ISAM simulator will create intelligent queries to request only a small piece of the table each time, as needed by the application. This is the recommended choice when working with huge tables, but it is slower than Cache Workareas when used with smaller tables.

  Note: Historic data can be handled only in ISAM Simulator mode.

You can determine how the RDD manages the tables with a callback code block. You can set this codeblock with the function:

SR_SetTableInfoBlock(<bInfoBlock>)

Usually, you do not need to set this code block in your application. SQLRDD default codeblock fits good most of the cases.

<bInfoBlock> is a codeblock used to retrieve the table information. When you execute the dbUseArea() function, SQLRDD calls this codeblock passing the table name as parameter. The codeblock must return an array the following information, defined in sqlrdd.ch:

```
#define TABLE_INFO_SIZE              16

#define TABLE_INFO_TABLE_NAME         1
#define TABLE_INFO_FILTERS            2
#define TABLE_INFO_PRIMARY_KEY        3
#define TABLE_INFO_RELATION_TYPE      4
#define TABLE_INFO_OWNER_NAME         5
#define TABLE_INFO_ALL_IN_CACHE       6
#define TABLE_INFO_CUSTOM_SQL         7
#define TABLE_INFO_CAN_INSERT         8
#define TABLE_INFO_CAN_UPDATE         9
#define TABLE_INFO_CAN_DELETE        10
#define TABLE_INFO_HISTORIC          11
#define TABLE_INFO_RECNO_NAME        12
#define TABLE_INFO_DELETED_NAME      13
#define TABLE_INFO_CONNECTION        14
#define TABLE_INFO_QUALIFIED_NAME    15
#define TABLE_INFO_NO_TRANSAC        16
```

A possible value for bInfoBlock is:

```
{ |cTableName| { upper(cTableName), {}, "", TABLE_INFO_RELATION_TYPE_SELECT, "", .F., "",
.T., .T., .T., .F. , “RECNO_”,  “DELETED_”} }
```

The default bInfoBlock from SQLRDD translates the file name’s attributes like “\/:.” Into underscores. Example:

"c:\data\accounts\chart.dbf"

Will be translated to:

"DATA_ACCONTS_CHART_DBF"

Note: See in APPENDIX 1 the full source code from this routine.

The exact use of each information array position is described below:

TABLE_INFO_TABLE_NAME

Returns the table name in this position. It is very useful when migrating from DBF. Your code may translate, as an example:

“c:\company1\products.dbf” to “COMPANY1_PRODUCTS”

The default behavior removes drive letters, pathnames and file extensions from the table name parameter as shown above.

TABLE_INFO_FILTERS

This position may hold an array with several filter expressions. The filter expression may be macro-executed in run time depending on the SR_EvalFilters( ) setup and results in a SQL expression to be added to the WHERE clause of the SQL Statements.

If you execute SR_EvalFilters(.T.) the expressions will be macro-executed. The default value in SQLRDD is FALSE.

As an example, an expression in TABLE_INFO_FILTERS array:

“customer like ‘“ + alltrim(M->cCustomer) + “’%”

Assuming that cCustomer has the value “ACM”, after the RDD internal macro execution of the filter expression (SR_EvalFilters(.T.)), we will have:

customer like ‘ACM%’

When we have more than one table in query (when we use SetJoin() function), SQLRDD needs to specify the table to be used by the filter expression. To solve this problem, you must always use the <ALIAS> argument inside the expression. So the correct expression is:

“<ALIAS>.customer like ‘“ + alltrim(M->cCustomer) + “’%”

The SQLRDD engine will translate the <ALIAS> argument into the correct alias name before sending the statement to the database server:

SELECT * FROM CUSTOMER A WHERE A.customer like ‘ACM%’

Note: You can assign several filter expressions in this TableInfo position using a nested array. All the expressions will be added to the WHERE clause of the generated SQL statement:

```
{“<ALIAS>.customer like ‘“ + alltrim( M->cCustomer ) + “’%”, “<ALIAS>.value IS NOT NULL”}
```

The SQL expression will result in:

*SELECT * FROM CUSTOMER A WHERE A.customer like ‘ACM%’ AND A.value IS NULL*

Default value for SQLRDD is {} in this position.

TABLE_INFO_PRIMARY_KEY

This position is used to inform the RDD the primary key column to be used by the Historic Data filters. Using historic data you get track of table updates and inserts in the time line, and the actual date determines the visible record.

Default value for SQLRDD is NIL in this position.

TABLE_INFO_RELATION_TYPE

This position describes how the RDD will handle the relations via SetJoin() function:

```
#define TABLE_INFO_RELATION_TYPE_SELECT             0
#define TABLE_INFO_RELATION_TYPE_JOIN               1
#define TABLE_INFO_RELATION_TYPE_OUTER_JOIN         2
```

Using TABLE_INFO_RELATION_TYPE_SELECT, the RDD will execute a SELECT in the child workarea to obtain the related record to each record in the main workarea. This is the default method used by all of the RDDs in xBase, but it is slower than the other methods available in SQLRDD

With TABLE_INFO_RELATION_TYPE_JOIN, the RDD will execute one single SELECT to get the lines to both workareas (main and child workareas). This is the best performance option, but it is not 100% compatible with other RDDs, because if the child record does not exist in a relation, the main record will **not be** shown as well. So use this option when you are completely sure of the relation integrity.

With TABLE_INFO_RELATION_TYPE_ OUTER _JOIN, the RDD will execute one single SELECT to get the lines to both workareas (main and child workareas). This is the second best performance option. Use this option when you are **not** completely sure of the relation integrity.

Default value for SQLRDD is TABLE_INFO_RELATION_TYPE_SELECT in this position.

TABLE_INFO_OWNER_NAME

This position is used to inform the table user name or schema. See the RDBMS documentation to learn how this feature can be used.

Default value for SQLRDD is “” in this position.

TABLE_INFO_ALL_IN_CACHE

When you pass .T. in this position you tell the RDD to use cache workarea. With .F., ISAM Simulator will be used.

Default value for SQLRDD is .F. in this position.

TABLE_INFO_CUSTOM_SQL

In this position you can specify a custom SQL command to be used as the table name. Updates, Deletes and Inserts neither are nor allowed when you use this feature.

TABLE_INFO_CAN_INSERT, TABLE_INFO_CAN_UPDATE and TABLE_INFO_CAN_DELETE

These options set the user rights for this workarea, with .T. or .F. in each position.

Default value for SQLRDD is .T. in these positions.

TABLE_INFO_HISTORIC

Use this position to enable the historic feature of the RDD.

Notes:

When you set this option .T., it is required to supply the position TABLE_INFO_PRIMARY_KEY with the table’s primary key;

When you set this option .T., it is required that the table was created with this option set, so the RDD also creates the specific control fields to the Historic Management.

Default value for SQLRDD is .F. in this position.

*Note: This feature can not be used when **TABLE_INFO_ALL_IN_CACHE** Is set to .T..*

TABLE_INFO_RECNO_NAME

Use this option to set a different name to the Recno() column. This column will be internally used by the system to store the physical record number.

Default value for SQLRDD is “SR_RECNO” in this position.

TABLE_INFO_DELETED_NAME

Use this option to set a different name to the Deleted() column. This column will be internally used by the system to store the physical deleted() status.

Default value for SQLRDD is “SR_DELETED” in this position.

TABLE_INFO_CONNECTION

Use this option to set a different connection object to this table. Connection Object can be obtained by the SR_GetConnection() function.

The SQL Parser

Using the Connection Classes, you can send any command direct to the database system.

But unfortunately, the SQL Syntax from the different database vendors is quite different. These differences are most evident when comparing:

- Date strings
- Joins and outer joins
- Sub queries and nested queries
- Identifiers
- Row / Table locking
- Transaction control
- Optimizations

It means that you normally have to use different SQL syntax with every target database, which becomes a hurdle when you want to have a portable application.

With SQL Parser, all you need to get portable with the major database systems is to use the xHarbour SQL Language. The parser “understands” the xHarbour SQL Syntax and generates an intermediate code, which can later be used by the Code Generator to automatically create the specific syntax supported by the SQL Server in use. This Parser is implemented with a Bison parser, and it means a very fast and reliable routine.

The SQL Parser supports SELECT, INSERT, UPDATE and DELETE commands. See the reference guide for the complete xHarbour SQL Syntax.

An xHarbour SQL command usually looks like this:

SELECT A.* FROM TABLE1 A WHERE A.ID = ? AND A.DATE = [20031231]

The “?” is a variable data parameter that will be supplied in the code generation phase. So you can compile the SQL command one time and execute it several times changing the parameters and consequently saving processor resources and gaining performance.

The function to compile a SQL command is:

SR_Parse(<cCommand>, [<@cErrorMessage>], [<@nErrorPosition>]) => apCode

The Code Generator

This component generates the SQL statement from the pCode created with the SQL Parser's SqlParse() function. The main function to generate the SQL code is:

SR_SQLCodeGen(<apCode>, [<aParamList>], <nSystemId>) => cSQLCode

You can pass parameters to the code generator function, when required by the SQL Command:

```
Local apCode
Local cCommand
Local cResult

cCommand := "SELECT A.* FROM TABLE1 A WHERE A.ID = ? AND "
cCommand += "A.NUM = ?"
apCode  := SR_SQLParse( cCommand )
cResult := SR_SQLCodeGen( apCode, {"100", 2}, SYSTEMID_MSSQL7 )
```

The resulting output will be:

SELECT A.* FROM TABLE1 A WITH (NOLOCK) WHERE A.[ID] = '100' AND A.[NUM] = 2

You can submit the resulting query to the database server and obtain the result set with the Connection Classes and their helper functions. Please check the reference guide to the complete

SQLRDD Features

Deleting records

SQLRDD fully supports records “marked for deletion” using an auto-created field named “DELETED” to store the status. This column name may be changed in each workarea with the TABLE_INFO_DELETED_NAME information array position or globally defined with the SR_DeletedName() function.

Your application can access the delete status directly with WORKAREA->SR_DELETED or with the deleted() function.

The whole dbDelete() behavior can be changed with the SR_UseDeleteds(lUseDeleteds) function. If you pass a .F. parameter, when you delete a record, it is gone forever, unless you have started a transaction control. The deleted() function and PACK command have no use in this situation, and when you delete a record, SQLRDD tries to move to the previous record in current order (dbskip(-1)).

Record numbering

SQLRDD uses an auto-created field named “SR_RECNO” to hold the record number. The method for filling this field depends on the target database system, but usually works with IDENTITY or ENUM.

Under Oracle, SQLRDD automatically creates a sequence SQ_NRECNO to control the record number. For another databases who have functionalities like IDENTITY columns, this is the chosen method.

Your application can access the record number directly with WORKAREA->SR_RECNO or with the Recno() function.

In SQLRDD, record numbers never change and their number can be higher than the quantity of records in the table. The PACK command has no meaning in SQLRDD if SR_UseDeleteds(lUseDeleteds) is set to false.

Supported data types

SQLRDD supports the same data types as DBFNTX RDD.

SQLRDD internally translates the xBase data types (CHARACTER, DATE, MEMO, NUMERIC and LOGICAL) into SQL data types.

The data types used internally by any given SQL system might vary. SQLRDD automatically chooses the best matching data type available in a given RDBMS.

Data Types Serialization

MEMO Fields can hold any data type, using automatic serialization. It means you can save Arrays, Objects, Hashes, Dates and Numbers in memo fields without any explicit conversion. Example:

```
DbCreate( “mytable”, {{“F1”,”M”,10,0},{“F2”,”M”,10,0},{“F3”,”M”,10,0}}, “SQLRDD” )
USE mytable
Append blank
Replace F1 with { 1,2,3,”abc”, date() }
Replace F2 with tbrowse():new()
Replace F3 with hashnew()
dbCLoseArea()

...

USE mytable VIA “SQLRDD”

? valtype( mytable->F1 )             => “A”
? valtype( mytable->F2 )             => “O”
? valtype( mytable->F3 )             => “H”

? F1[4], F1[2]                       => “abc”  2
? F2:className()                     => “TBROWSE”
```

Note for Oracle users: Memo fields under Oracle using ODBC connection are limited to 4000 bytes (VARCHAR2 datatype). Using Oracle Native Access you will not experience such problem.

Index Management

SQLRDD fully supports complex index expressions, index bag and tag names, in the exact same fashion as DBFCDX RDD. Drive letters, path names and file extensions are internally eliminated from the index name, and are limited to 30 characters.

Each time you create a complex index expression, SQLRDD automatically adds a new column in your table to store the index expression result. These columns are named “INDKEY_001… INDKEY_999”, and the expression result is limited in 256 characters long. The INDKEY_nnn field contents is automatically maintained wherever the corresponding index is in use at the moment or not. This kind of index is named “Synthetic Index” in SQLRDD.

You may want to use Synthetic Indexes practice even with regular indexes when working with very large tables in databases like Postgres and Oracle. Benefits are best SKIP, SEEK, GOTOP and GOBOTTOM performance with very large tables. Disadvantages are waste of physical space and prohibition to UPDATE or INSERT in table from outside

xHb application. You can turn this feature on by adding following line BEFORE creating the Index:

SR_SetSyntheticIndex(.T.) ➔ lOldSet

Note: Index key expression length may be limited by database. Firebird has the lowest limit we could find, so you should try to use small index keys when possible.

All of the index information is stored in a auto-created table named "SR_MGMNTINDEXES". You should NEVER change the contents of this table because SQLRDD depends on this information for internal system management.

Additionally, SQLRDD introduces a new concept for using indexes, based on the SQL engine capabilities:

In SQL Database Servers, indexes are used in general to search data, and **not** for ordering. All of the modern database systems have an extreme optimizer and statistics engine, and this is responsible for choosing **what index will be used or not used in each query**. This can be reviewed in any RDBMS that shows the "Execution Plan". Thus, in SQLRDD, seeking data by any index is not a guarantee that the RDBMS will internally use this index.

Indexes in SQLRDD can be used to declare the fields that will be used in the resulting SQL sentence when you use a dbSeek() function, and for sorting the result set. **It is not related to the index physical existence**. But if an index with the same fields used in a query exists in the database, you have better chances to work with acceptable performance.

Examples:

1 – Creating a physical index in the database system:

```
/* CODE_ID and DESCR are character, DAYS is numeric (3) and DATE_LIM date */

USE TEST_TABLE SHARED VIA "SQLRDD" NEW

Index on CODE_ID+DESCR to TEST_TABLE_IND01
Index on DAYS+DATE_LIM to TEST_TABLE_IND02
```

This sample illustrates how to create a physical index in any database system. Note that it is not necessary to convert any data type. Functions like dbCreateInd() can be used as well as commands.

2 – Using the indexes:

```
/* CODE_ID and DESCR are character, DAYS is numeric (3) and DATE_LIM date */

USE TEST_TABLE SHARED VIA "SQLRDD" NEW
```

```
SET INDEX TO ("CODE_ID")
SET INDEX TO ("DAYS+DATE_LIM") ADDITIVE
SET INDEX TO ("DAYS+CODE_ID") ADDITIVE
…
SET ORDER TO 1
SEEK “0233”
…
SET ORDER TO 2
SEEK str( nDays, 3 )  + dtos( date() )
…
SET ORDER TO 3
SEEK( str( nDays, 3 ) + “0233”
…
```

Notes about this sample:

- In SQLRDD, indexes can be invoked by its name or by its expression
- You can use functions in index expressions, but SQLRDD will criate a column in the table to store the resulting index expression.
- You can mix data types in index expressions without having to use type conversion functions
- You can specify an index that does not exist physically (3rd index), but it can be low in performance

Transactional Control and Locking

A transaction is a peace of code where all database changes can be accepted or reverted as a block. In other words, we have can guarantee the integrity of interdependent changes on the database.

To start a transaction block, you simply issue a SR_StartTransaction(). When transaction block ends, you just issue SR_CommitTransaction() to confirm changes or SR_RollBackTransaction() to revert all changes since transaction started.

Important Notes:

- SQLRDD acts to standardize transaction behavior in all supported databases, so the final transaction control relies on target RDBMS resources.
- If workstation is turned off during a transaction block, RDBMS will Roll Back all changes automatically.
- All changed and inserted lines remains locked until the end of the transaction. All dbCoommit() calls are ignored inside transaction blocks

- Transactions opened inside transactions (nested transactions) have no practical effect. In other words, if you issue SR_StartTransaction() twice, commit will occur only in second SR_CommitTransaction() call.

- Any DML command (like dbCreate(), INDEX ON…, etc.) will automatically end any open transaction and release all locks.

Tracing SQL calls

SQLRDD has 3 logging options:

- Tracelog: Trace to a dbf file named SQLLOG.DBF all the SQL commands sent to the database system. This is very useful for debugging purposes. To start and stop tracing, use:

  SR_StartLog([<nConnection>])
  SR_StopLog([<nConnection>])

  Note: These functions must be called after SR_AddConnection()

- TimeTrace: Trace to a dbf file named LONG_QRY.DBF all the SQL commands sent to the database system that takes more than 1000 milliseconds to be executed by the RDBMS. This is very useful for fine tuning your application. To change the minimum trace time, use:

  SR_SetTimeTrace([<nConnection>], [<nMilisseconds>]) => nOldValue

- ScreenTrace: Display each SQL command to screen before issuing it to database, using Alert() function. To start and stop tracing, use:

  SR_StartTrace([<nConnection>])
  SR_StopTrace([<nConnection>])

SQLRDD also creates some additional log files:

sqlerror.log	Logs all failed SQL sentences and all failed connections
<databaseName>.log	Logs database-specific erros codes and events

SQLRDD Version Control

The SQLRDD version information is stored in a auto-created table named “SR_MGMNTVERSION”. You should NEVER change the contents of this table because SQLRDD depends on this information for internal system management.

To check for the current SQLRDD linked version, as well as the current connected database, use the following code:

```
/*----------------------------------------------------------------------*/

#include "sqlrdd.ch"
#include "dbinfo.ch"

REQUEST SQLRDD
REQUEST SR_ODBC

FUNCTION MAIN(cDsn)

   Local nCnn, oSql, i

   If empty( cDsn )
      Return NIL
   EndIf

   nCnn := SR_AddConnection( CONNECT_ODBC, cDSN )

   If nCnn < 0
      ? "Connection error. See SQL1.LOG for details."
      Return NIL
   EndIf

   use TEST_TABLE3 new via "SQLRDD"

   i := select()

   ? "Workarea number :", i
   ? "RDD Version     :", dbInfo( DBI_RDD_VERSION )
   ? "Host Database   :", dbInfo( DBI_DB_VERSION ) // see sqlrdd.ch for details
   ? ""

Return

/*----------------------------------------------------------------------*/
```

SQLRDD Language Support

You can use SR_SetBaseLang(nLang) to change the SQLRDD message's language. nLang may be:

```
MSG_EN       1            (English - default)
MSG_PT       2            (Brazilian Portuguese)
```

You can write your own language messages redefining the SR_Msg(nMsg) function. The SR_Msg(nMsg) source code is supplied in APPENDIX 2.

Compiling and linking the application

To compile, link and execute SQLRDD in your programs, follow these steps:

1. Add sql.lib to your link script or project in xBuild

2. Add sqlrdd.ch in the beginning of the source code files

3. In the first PRG (Main function) you should declare:

   REQUEST SQLRDD

   And also declare one or more from the following connection classes:

```
REQUEST SR_ODBC           /* ODBC Connection */
REQUEST SR_MYSQL          /* My SQL 4.1 direct connection */
REQUEST SR_PGS            /* Postgres >= 7.3 native connection */
```

   Notes:

   When using **Postgres Native Support**, you should add **pgs.ch** to your first PRG file and libpq.dll to system path. These files are shipped with SQLRDD and you can find then in xHb\dll and xHb\lib folders.

   When using **MySQL Native Support**, you should add **mysql.ch** to your first PRG file and libmysql.dll to system path. These files are shipped with SQLRDD and you can find then in xHb\dll and xHb\lib folders.

   Under Windows, the xBuild utility will automatically add libpq.lib or libmysql.lib to your link script when it finds the includes (pgc.ch or mysql.ch) in your sources.

4. When opening tables, inform the SQLRDD driver:

```
USE TABLE1 VIA “SQLRDD” NEW

or

dbUseArea( .T., “SQLRDD”, “TABLE1” )

or

RddSetDefault( “SQLRDD” )
```

Performance tuning tips

The following tips are very useful to have your application running with acceptable performance:

1. Use correct, nom repetitive indexes. Keep in mind that every index consumes performance when adding records. So if you already have an index on COLUMN1 + COLUMN2, another index on COLUMN1 is useless.

2. Avoid using index expressions containing User Defined Functions (UDFs) or other functions except DTOS() and STR(). UDF Indexes are also slow when indexing. This is not a problem.

3. Filter expressions to be solved in “client” side, such as “SET FILTER” and “DELETED() records” can also slow down your application. Try to change filter routines to SET SCOPE. The filter is solved at server side and it is index optimized.

4. Open your tables in SHARED mode. EXCLUSIVE mode requires much more system resources and should be used only under extreme circumstances. To be 100% sure you are not using EXCLUSIVE table open, issue the following command:

   ```
   SR_SetFastOpen(.T.)
   ```

   Note: This is the default SQLRDD set.

5. When opening a table, SQLRDD automatically opens its indexes, as DBFCDX does. So if you manually open the indexes, the same operation will be performed two times. This default condition can be changed with:

   ```
   SET AUTOPEN OFF
   ```

6. Routines where you need to delete several records can be greatly optimized with the use of SQL sentences.

   Example:

```
-----------------------------------------------------------------

USE MyTable.dbf ALIAS MyAlias
SELECT MyAlias
SET ORDER TO MyIndex
SEEK Key + Key2

WHILE MyAlias->col1 = Key .and. MyAlias->col2 = Key2
   IF dbRlock()
      dbDeleted()
      dbRUnlock()
   ENDIF
   SKIP
ENDDO

IF fLock()
   PACK
ENDIF

-----------------------------------------------------------------
```

   Can be optimized with the cross platform solution below:

```
-----------------------------------------------------------

local apCode, oSql
apCode := SR_SQLParse( “delete from mytable_dbf where col1 = ? and col2 = ?”)
oSql   := SR_GetConnection()
oSql:exec( SR_SQLCodeGen( apCode, { Key, Key2 }, oSql:nSystemID ) )

-----------------------------------------------------------
```

   Note: Same code runs with all supported SQL databases with no change.

7. Routines where you need to apply a value to one or more columns in several records can be greatly optimized with the use of SQL sentences.

   Example:

```
-------------------------------------------------------------

USE MyTable.dbf ALIAS MyAlias
SELECT MyAlias
SET ORDER TO MyIndex
SEEK Key + Key2

WHILE MyAlias->col1 = Key .and. MyAlias->col2 = Key2
   IF dbRlock()
      REPLACE col3 with 0
      REPLACE col4 with date()
      dbRUnlock()
   ENDIF
   SKIP
ENDDO

-------------------------------------------------------------
```

   Can be optimized with the cross platform solution below:

```
-----------------------------------------------------------

local apCode, oSql
apCode := SR_SQLParse( “update mytable_dbf set col3 = 0, col4 = ? where col1 = ?
and col2 = ?”)
oSql   := SR_GetConnection()
oSql:exec( SR_SQLCodeGen( apCode, { date(), Key, Key2 }, oSql:nSystemID ) )

-----------------------------------------------------------
```

   Note: Same code runs with all supported SQL databases with no change.

8 – Synthetic Indexes: Synthetic indexes are those created by adding a hidden column in table to hold index key expression return value (See Index Management chapter for a better explanation about this method). Setting synthetic index on, all orders are created this way. Benefits are best SKIP, SEEK, GOTOP and GOBOTTOM performance with very large tables. Disadvantages are waste of physical space and prohibition to UPDATE or INSERT in table from outside xHb application. Anyway, you can browse a million records table as fast as a 200 records table.

This speed gain is not noticeable in all databases. Please avoid using Synthetic Indexes with other databases than Postgres and Oracle.

Compatibility and Limitations

SQLRDD is a RDD almost compatible with DBFCDX. Here we have a list of know compatibility and limitation issues regarding SQLRDD:

- Any DML command (like dbCreate(), INDEX ON…, etc.) will automatically end any open transaction and release all locks.

- If you choose to use Firebird, please check for index key size limit. It's a known Firebird bug.

- You can not use another workarea's field as an index expression in SQLRDD

APPENDIX 1:

Source code to SQLRDD's default bInfoBlock callback function

```
/*----------------------------------------------------------------*/

#include "hbsql.ch"
#include "error.ch"
#include "sqlrdd.ch"
#include "msg.ch"

Function SR_TableAttr( cTableName )

   /* Translates "c:\data\accounts\chart.dbf" to "DATA_ACCONTS_CHART" */

   local aRet, cOwner := ""

   if cTableName[2] == ":"
      /* Remove drive letter */
      cTableName := SubStr( cTableName, 3 )
   endif

   cTableName := strtran( alltrim(lower(cTableName)), ".dbf", "_dbf" )
   cTableName := strtran( cTableName, ".ntx", "" )
   cTableName := strtran( cTableName, ".cdx", "" )
   cTableName := strtran( cTableName, "\", "_" )
   if cTableName[1] == "/"
      cTableName := SubStr( cTableName, 2 )
   endif
   cTableName := strtran( cTableName, "/", "_" )
   cTableName := strtran( cTableName, ".", "_" )
   cTableName := alltrim( cTableName )

   if len( cTableName ) > 30
      cTableName := SubStr( cTableName, len( cTableName ) - 30 + 1 )
   endif

   cOwner := SR_SetGlobalOwner()
   If (!Empty(cOwner)) .and. cOwner[-1] != "."
      cOwner += "."
   EndIf

   aRet := { upper(cTableName),;
             {},;
             "",;
             TABLE_INFO_RELATION_TYPE_OUTER_JOIN,;
             SR_SetGlobalOwner(),;
             .F.,;
             "",;
             .T.,;
             .T.,;
             .T.,;
             .F.,;
             ,;
             ,;
             ,;
             cOwner + upper(cTableName) }

Return aRet

/*----------------------------------------------------------------*/
```

APPENDIX 2:

Source code to SQLRDD's default language support (utilslang.prg).

```
/*
* SQLRDD Language Utilities
* Copyright (c) 2003 - Marcelo Lombardo  <marcelo@xharbour.com.br>
* All Rights Reserved
*/

Static nMessages   := 28

GLOBAL nBaseLang   := 1
GLOBAL nSecondLang := LANG_EN_US
GLOBAL nRootLang   := LANG_PT_BR    // Root since I do it on Brazil

Static aMsg1 := ;
{ ;
   "Attempt to write to an empty table without a previous Append Blank",;
   "Undefined SQL datatype: ",;
   "Insert not allowed in table : ",;
   "Update not allowed without a WHERE clause.",;
   "Update not allowed in table : ",;
   "Invalid record number : ",;
   "Not connected to the SQL dabase server",;
   "Error Locking line or table (record# + table + error code):",;
   "Unsupported data type in : ",;
   "Syntax Error in filter expression : ",;
   "Error selecting IDENTITY after INSERT in table : ",;
   "Delete not allowed in table : ",;
   "Delete Failure ",;
   "Last command sent to database : ",;
   "Insert Failure ",;
   "Update Failure ",;
   "Error creating index : ",;
   "Index Column not Found - ",;
   "Invalid Index number or tag in OrdListFocus() : ",;
   "Seek without active index",;
   "Unsupported data type at column ",;
   "Unsupported database : ",;
   "Could not setup stmt option",;
   "RECNO column not found (column / table): ",;
   "DELETED column not found (column / table): ",;
   "Invalid dbSeek or SetScope Key Argument",;
   "Error Opening table in SQL database",;
   "Data type mismatch in dbSeek()" ;
}

Static aMsg2 := ;
{ ;
   "Tentativa de gravar um registro em tabela vazia sem antes adicionar uma linha",;
   "Tipo mde dado SQL indefinido : ",;
   "Inclusão não permitida na tabela : ",;
   "Update não permitido SEM cláusula WHERE.",;
   "Alteração não permitida na tabela : ",;
   "Número de Registro inexistente :",;
   "Não conectado ao servidor de banco de dados",;
   "Erro travando registro ou tabela (num.registro + tabela + cod.erro):",;
   "Tipo de dado não suportado em : ",;
   "Erro de sitaxe em expressão de filtro : ",;
   "Erro ao selecionar IDENTITY após INSERT na tabela : ",;
   "Exclusão não permitida na tabela : ",;
   "Erro na exclusão ",;
   "Último comando enviado ao banco de dados : ",;
   "Erro na inclusão ",;
   "Erro na alteração ",;
   "Erro criando índice : ",;
   "Coluna de índice não existente - ",;
   "Indice ou tag inválido em OrdListFocus() : ",;
```

```
   "Seek sem índice ativo.",;
   "Tipo de dado inválido na coluna ",;
   "Banco de dados não suportado : ",;
   "Erro configurando stmt",;
   "Coluna de RECNO não encontrada (coluna / tabela): ",;
   "Coluna de DELETED não encontrada (coluna / tabela): ",;
   "Argumento chave inválido em dbSeek ou SetScope",;
   "Erro abrindo tabela no banco SQL",;
   "Tipo de dado inválido em dbSeek()";
}

Static aMsg3 := ;
{ ;
   "Attempt to write to an empty table without a previous Append Blank",;
   "Undefined SQL datatype: ",;
   "Insert not allowed in table : ",;
   "Update not allowed without a WHERE clause.",;
   "Update not allowed in table : ",;
   "Invalid record number : ",;
   "Not connected to the SQL dabase server",;
   "Error Locking line or table (record# + table + error code):",;
   "Unsupported data type in : ",;
   "Syntax Error in filter expression : ",;
   "Error selecting IDENTITY after INSERT in table : ",;
   "Delete not allowed in table : ",;
   "Delete Failure ",;
   "Last command sent to database : ",;
   "Insert Failure ",;
   "Update Failure ",;
   "Error creating index : ",;
   "Index Column not Found - ",;
   "Invalid Index numner or tag in OrdListFocus() : ",;
   "Seek without active index",;
   "Unsupported data type at column ",;
   "Unsupported database : ",;
   "Could not setup stmt option",;
   "RECNO column not found (column / table): ",;
   "DELETED column not found (column / table): ",;
   "Invalid dbSeek or SetScope Key Argument",;
   "Error Opening table in SQL database",;
   "Data type mismatch in dbSeek()";
}

Static aMsg4 := ;
{ ;
   "Attempt to write to an empty table without a previous Append Blank",;
   "Undefined SQL datatype: ",;
   "Insert not allowed in table : ",;
   "Update not allowed without a WHERE clause.",;
   "Update not allowed in table : ",;
   "Invalid record number : ",;
   "Not connected to the SQL dabase server",;
   "Error Locking line or table (record# + table + error code):",;
   "Unsupported data type in : ",;
   "Syntax Error in filter expression : ",;
   "Error selecting IDENTITY after INSERT in table : ",;
   "Delete not allowed in table : ",;
   "Delete Failure ",;
   "Last command sent to database : ",;
   "Insert Failure ",;
   "Update Failure ",;
   "Error creating index : ",;
   "Index Column not Found - ",;
   "Invalid Index number or tag in OrdListFocus() : ",;
   "Seek without active index",;
   "Unsupported data type at column ",;
   "Unsupported database : ",;
   "Could not setup stmt option",;
   "RECNO column not found (column / table): ",;
   "DELETED column not found (column / table): ",;
   "Invalid dbSeek or SetScope Key Argument",;
```

```
   "Error Opening table in SQL database",;
   "Data type mismatch in dbSeek()";
}

Static aMsg5 := ;
{ ;
   "Attempt to write to an empty table without a previous Append Blank",;
   "Undefined SQL datatype: ",;
   "Insert not allowed in table : ",;
   "Update not allowed without a WHERE clause.",;
   "Update not allowed in table : ",;
   "Invalid record number : ",;
   "Not connected to the SQL dabase server",;
   "Error Locking line or table (record# + table + error code):",;
   "Unsupported data type in : ",;
   "Syntax Error in filter expression : ",;
   "Error selecting IDENTITY after INSERT in table : ",;
   "Delete not allowed in table : ",;
   "Delete Failure ",;
   "Last command sent to database : ",;
   "Insert Failure ",;
   "Update Failure ",;
   "Error creating index : ",;
   "Index Column not Found - ",;
   "Invalid Index number or tag in OrdListFocus() : ",;
   "Seek without active index",;
   "Unsupported data type at column ",;
   "Unsupported database : ",;
   "Could not setup stmt option",;
   "RECNO column not found (column / table): ",;
   "DELETED column not found (column / table): ",;
   "Invalid dbSeek or SetScope Key Argument",;
   "Error Opening table in SQL database",;
   "Data type mismatch in dbSeek()";
}

Static aMsg6 := ;
{ ;
   "Attempt to write to an empty table without a previous Append Blank",;
   "Undefined SQL datatype: ",;
   "Insert not allowed in table : ",;
   "Update not allowed without a WHERE clause.",;
   "Update not allowed in table : ",;
   "Invalid record number : ",;
   "Not connected to the SQL dabase server",;
   "Error Locking line or table (record# + table + error code):",;
   "Unsupported data type in : ",;
   "Syntax Error in filter expression : ",;
   "Error selecting IDENTITY after INSERT in table : ",;
   "Delete not allowed in table : ",;
   "Delete Failure ",;
   "Last command sent to database : ",;
   "Insert Failure ",;
   "Update Failure ",;
   "Error creating index : ",;
   "Index Column not Found - ",;
   "Invalid Index number or tag in OrdListFocus() : ",;
   "Seek without active index",;
   "Unsupported data type at column ",;
   "Unsupported database : ",;
   "Could not setup stmt option",;
   "RECNO column not found (column / table): ",;
   "DELETED column not found (column / table): ",;
   "Invalid dbSeek or SetScope Key Argument",;
   "Error Opening table in SQL database",;
   "Data type mismatch in dbSeek()";
}

Static aMsg7 := ;
{ ;
   "Attempt to write to an empty table without a previous Append Blank",;
```

```
   "Undefined SQL datatype: ",;
   "Insert not allowed in table : ",;
   "Update not allowed without a WHERE clause.",;
   "Update not allowed in table : ",;
   "Invalid record number : ",;
   "Not connected to the SQL dabase server",;
   "Error Locking line or table (record# + table + error code):",;
   "Unsupported data type in : ",;
   "Syntax Error in filter expression : ",;
   "Error selecting IDENTITY after INSERT in table : ",;
   "Delete not allowed in table : ",;
   "Delete Failure ",;
   "Last command sent to database : ",;
   "Insert Failure ",;
   "Update Failure ",;
   "Error creating index : ",;
   "Index Column not Found - ",;
   "Invalid Index number or tag in OrdListFocus() : ",;
   "Seek without active index",;
   "Unsupported data type at column ",;
   "Unsupported database : ",;
   "Could not setup stmt option",;
   "RECNO column not found (column / table): ",;
   "DELETED column not found (column / table): ",;
   "Invalid dbSeek or SetScope Key Argument",;
   "Error Opening table in SQL database",;
   "Data type mismatch in dbSeek()";
};

Static aMsg

/*----------------------------------------------------------------------*/
INIT PROCEDURE SR_Init2

  aMSg := { aMsg1, aMsg2, aMsg3, aMsg4, aMsg5, aMsg6, aMsg7 }

RETURN
/*----------------------------------------------------------------------*/
```