Document describes a way of migrating CA-Clipper or (X)Harbour applications to an SQL environment using Mediator RDD. It also contains some hints how to effectively use Mediator and the description of additional functions of the software package.

Mediator
v.5.0

User's Guide and Reference

Contents

I. Introduction

The Mediator package enables CA-Clipper and (x)Harbour applications to cooperate with advanced RDBMS (*Relational DataBase Management System*). The *CA-Clipper* and *(x)Harbour* applications use a common data access interface: RDD. We will refer to the mentioned applications as *XBASE applications.* Methods of storage and data access used in the traditional XBASE application are not free from faults that make achievement of satisfactory reliability, security and efficiency impossible. Using a relational database managed by one of the modern, secure and reliable systems allows eliminating practically all problems associated with the usage of *.dbf* files and therefore leaves up to application only the functions related to the data manipulation and presentation. XBASE applications' architecture allows replacing a data access driver (RDD, *Replaceable Data Driver*). This architecture made it possible to create the Mediator package, which from the point of view of an application fulfills the RDD functionality and allows transparent access to the modern database in the same way as it was possible for *.dbf files (see Figure 1).

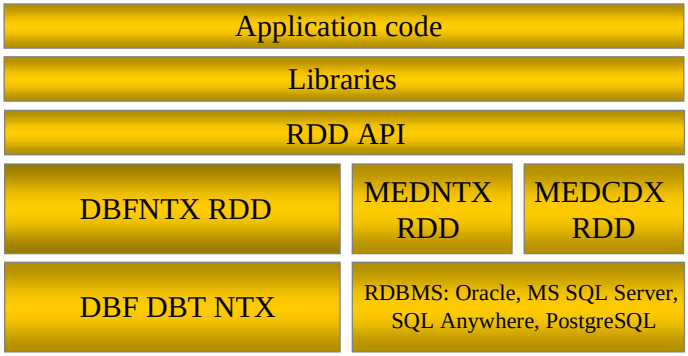


Figure 1. Architecture of XBase/Mediator application

1. MEDIATOR

Mediator is an intermediate layer between an *xBase* application and a database server, and consists of four basic modules (see Figure 2):

1. **RDD module** – the RDD specification compliant driver which practically eliminates the need of modification of an application using it.
2. **Network client** – the module creating the connection to an agent running on server. It uses one of the popular network protocols (TCP/IP or SPX/IPX). To

minimize network traffic, information exchange is organized according to the specially designed protocol, which specifies the rules of data fragmentation and encoding.

3. **Communication server** – the multithreaded server that receives queries and sends answers to a client application using the same protocol as the one used by the network client.
4. **Command server** – the module analyzing commands received from a client machine. If the command requires querying the database, a query legible for RDBMS is formed and sent using API appropriate for the server used. Results of the query are sent back to the client via communication server.

The server works continuously. It is constantly prepared to receive commands from the workstation. If a connection request from a new station is received, a new thread, dedicated to serving the new workstation is created. The thread is working until the receipt of a disconnect command or detection that the workstation stopped working. In both cases the resources used by the client are freed, including all database and record locks. All uncommitted transactions are rolled back.

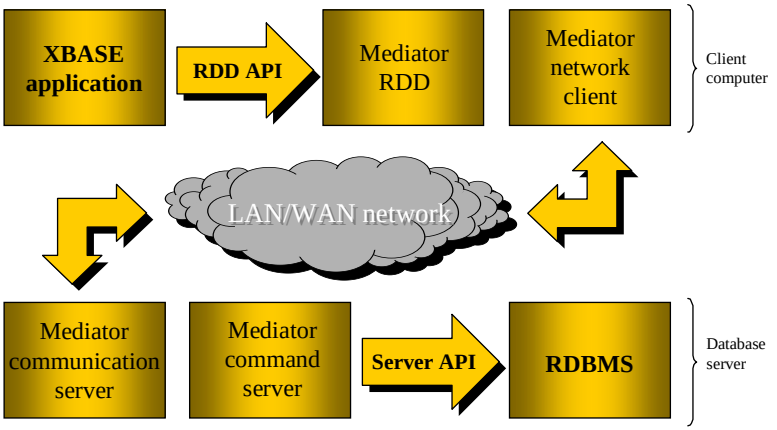


Figure 2. Mediator software architecture

The Mediator package contains a library to be linked to XBASE applications as well as an agent installed on Windows NT/2000/XP, Linux or SUN SPARC Solaris platform (depending on Mediator package version). The agent is responsible for the communication with the RDBMS. The RDBMS can be installed on the same computer as the Mediator agent or it can work on another machine as well (Figure 3). Mediator communicates with the server through the server-specific API. CA-Clipper applications linked with Mediator libraries can work on PC computers with DOS operating systems or in a Windows 95/98/NT/2000/XP DOS console window.

(x)Harbour applications using the Mediator library can run on standard PC computers with the Windows 95/98/Me/NT/2000/XP operating system installed.

For the purpose of communication between application and agent, the TCP/IP or SPX/IPX protocol is used. The Mediator library contains two RDD drivers called MEDNTX and MEDCDX. These drivers are compliant with Clipper DBFNTX and DBFCDX drivers respectively and they are dedicated to the fast porting of an *xBase* application to the RDBMS environment. Additionally, Mediator libraries contain functions which allow direct usage of SQL statements. Both drivers can be used simultaneously and do not exclude using the original DBFNTX and DBFCDX drivers. It means that within a single application some tables dedicated to the *xBase* application can be opened in the mode of compliance with the Clipper (MEDNTX driver), some other tables can be shared with client-server applications in the SQL mode, and other, for example, temporary or configuration data can be stored in *.dbf* files. It is a very useful feature of the system, especially when many temporary tables not needed by other users are created. Temporary local files usually do not pose danger to the data security, while they significantly reduce network traffic (as long as they are created on local disks) and they do not cause excessive fragmentation in RDBMS.

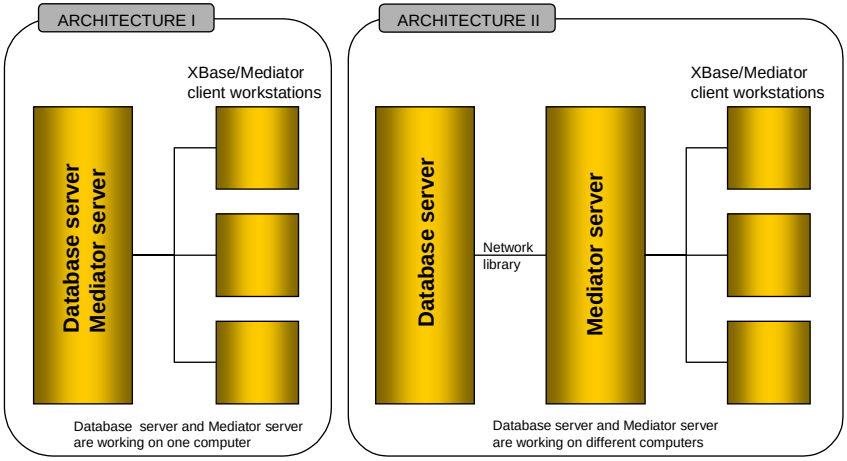


Figure 3. Hardware architectures for Mediator/database software

2. The MEDNTX driver

The MEDNTX driver is compliant with CA-Clipper DBFNTX driver. It was designed to minimize the amount of changes in a ported application. The driver emulates the *xBase* application behavior with regard to the *RECNO()* record numbers, marking records for deleting (*DELETED()*), locking records and tables as well as working on the complex index expressions. During the first connection to the database the driver creates a repository on user's account, which is used during all the time of application activity. The repository holds the information about all used tables together with the description of their structures and additional information necessary for keeping maximum compatibility with the *xBase* applications and DBFNTX. The drawback resulting from achieving this compatibility is a certain increase in difficulty of cooperation with other applications created by the client-server tools and using the shared data. The MEDNTX driver has a built-in interface for database transaction system as well as many extensions enabling the use of SQL language in the application source code.

3. The MEDCDX driver

The MEDCDX driver is compliant with DBFCDX standard. The driver contains all features of the MEDNTX driver. Additionally, implementation of handling indexes is compliant with the DBFCDX driver. Both drivers (MEDNTX and MEDCDX) can use the scope mechanism (SCOPE). The same library contains functions of the MEDNTX driver and functions of the MEDCDX driver (MEDNTXnn.LIB). In case of CA-VO the functions of the MEDCDX driver are implemented in the MEDCDX.RDD library.

4. Application migration – the MEDNTX and MEDCDX drivers

It is simple to adjust an application to work with Mediator. It is enough to add a Mediator header file to every source file and change the default DBFNTX (DBFCDX) driver to the MEDNTX (MEDCDX) driver with one function call. In case of CA-Clipper or (x)Harbour applications, the object code is linked with the delivered library after the compilation. The *.exe* file contains the application ready for the cooperation with RDBMS. In case of CA-VO applications, linking is not necessary as the libraries are supplied as dynamic link libraries. The next step is to move data and indexes from *.dbf*, *.ntx* and *.cdx* files to the database server. Remember that using the functions that operate directly on the filesystem (i.e. FILE(), ADIR()) is not applicable for the structures stored in RDBMS. This is why it is necessary to replace them with calls of the equivalent Mediator functions working on RDBMS tables.

5. Language extensions

The process of adjusting applications for cooperation with RDBMS is a little more complicated, if the programmer wants to use the available language extensions. The Mediator offers several extension groups:

- An interface for a transaction system (`BEGIN TRANSACTION, COMMIT TRANSACTION, ROLLBACK TRANSACTION` commands),
- SQL queries sent from the XBASE applications (`USE` *`V_TAB`* `AS SELECT` ... command) -- *V_TAB* is a virtual table, the rows of which are the records produced as a result of the query – it can be browsed with the use of standard XBASE instructions.
- Calling procedures stored in the database server,
- Filters set on server and defined by means of SQL conditions,
- About 140 additional functions which help in exploiting the RDBMS capabilities.

Using the extensions requires basic knowledge about mechanisms of RDBMS as well as about the basics of SQL. In exchange, the programmer is able to choose from a wide range of techniques that improve the performance of application. Introduction of transactions ensures logical consistency of data and prevents the whole system from stopping because of the breakdown of one of the workstations. The direct application of the SQL queries as well as procedures built into the database server allows to transfer the majority of complicated calculations onto the server. Therefore, the investment related to the expansion of workstations can be minimized, network traffic is significantly reduced and overall system efficiency is increased.

6. Working in WAN

Thanks to the significant reduction of network traffic, the Mediator makes the operation of XBASE applications better in the wide area network. A special method of data encoding limits the network traffic to minimum. It is several times lower compared to a traditional system. In order to improve the work in WAN further, a special software called Terminal, produced by OTC, can be used. This software allows for the terminal work of the CA-Clipper application. The application server is a Windows NT/2000/XP system and any PC computer can work as the terminal. Many stations can effectively work via low throughput link with the use of the Terminal software.

7. Summary

The Mediator package offers a simple way of application migration from the XBASE to the RDBMS environment. After transfer, applications using the Mediator RDD can work on data stored in RDBMS sharing it with the graphic client-server modules

developed with any SQL tools. Therefore, existing systems can be migrated, module by module, with the use of any tool designed for creating client-server applications. Systems built from the beginning with client-server tools gain the character user interface, which is not available in many modern database management systems. New modules can cooperate with old ones *on-line*, or they can receive and process data from the old application in batches. Such process can be made completely automatic, and old and new data can be processed and presented with available SQL tools (for example, OLAP or data warehouses). Besides, the user is able to access all tools and capabilities offered by a database server - mainly those securing data against the loss or unauthorized access. In case of the hardware failure, the archiving mechanisms as well as transaction journals created by the server enable quick resuming of work after failure without the loss of information.

II. Getting started

1. The necessary components

The Server

Operating system

- Windows NT/2000/XP server
  The NWLINK IPX/SPX Compatible Transport protocol or TCP/IP protocol is necessary to be installed on Windows server Make sure the latest service pack is also installed.

or

- Sun Solaris version required for installing Oracle

or

- Linux, version appropriate for installing Oracle or PostgreSQL (7.2 or newer) or MySQL (4.0 or newer).

Database:

Depending on the Mediator version it can be the Oracle server, Microsoft SQL Server (version 7.0 or later), Sybase Adaptive Server Anywhere (version 6.0 or later) or PostgreSQL (version 7.2 or later) or MySQL (version 4.0 or later).

- The Oracle Server.
  The Oracle Server needs to be up and running. In the tablespaces there should be enough free space to load data as well as to operate. The size of SGA should be adjusted to a number of users and expected load. Other recommended parameters:
  OPEN_CURSORS = 300
  There should be the Oracle user who will be the owner of objects (tables and indexes), with appropriate privileges for connecting to the database and creating objects.
- Microsoft SQL Server (version 7.0 or later)
  The server should be installed. During installation appropriate code page and sort order needs to be chosen. Create the database of sufficient capacity to store the data. Create user that is going to be the owner of the objects with permissions for connecting to the server and to write in that database (the database created should be the default database for that user). Configure ODBC system data source for the installed server.

- Sybase Adaptive Server Anywhere (version 6.0 or later)
  The server should be installed. Create the database of sufficient capacity to store the data. Create user that is going to be the owner of the objects with permissions for connecting to the server and to write and create objects in that database. Increase Max_cursor_count and Max_statement_count parameters for that database. Depending on the application it can be set to for example 300 or to 0 (no limits). Configure ODBC system data source for the installed server.
- PostgreSQL (version 7.2 or later)
  The server should be installed with the appropriate character set. The computer where Mediator server is installed should be able to connect to the PostgreSQL database (it could be done by editing pg_hba.cfg and adding the appropriate entry). Create the database where the data will be stored and the user – owner of created objects. It is absolutely necessary to disable full table scans (in the postgresql.conf file add the following entry: enable_seqscan=false). Install and configure ODBC for the created database.
- MySQL (version 4.0 or later)
  The server should be installed with the appropriate character set. The computer where Mediator server is installed should be able to connect to the MySQL database. Create the database where the data will be stored and the user – owner of created objects. Install and configure ODBC for the created database.

MEDIATOR
The MEDIATOR server should be installed and functional.

The Client

CA-Clipper client

- The IPX/SPX or TCP/IP protocol should be installed and functional. If the client is to communicate via IPX/SPX protocols, the following options need to be configured:
  - ◊ In DOS: NETX, VLM or 32-bit NetWare client
  - ◊ In Windows 95: Microsoft IPX/SPX Compatible Transport
  - ◊ In Windows 98: Microsoft IPX/SPX Compatible Transport
  - ◊ In Windows NT/2000/XP: NWLINK IPX/SPX Compatible Transport (remember to install **omvdd.dll** file in **winnt\system32** directory)

  Using IPX, take note of the client and server frame compatibility.

  If the client is using TCP/IP, one of the following protocols needs to be installed and functional:
  - ◊ In DOS :32-bit NetWare DOS client with TCP/IP
  - ◊ In Windows 95: Microsoft TCP/IP (if Winsock2 installed then remember to install **omvsx.vxd** file in **windows\system** directory)

- ◊ In Windows 98: Microsoft TCP/IP (remember to install **omvsx.vxd** file in **windows\system** directory)
- ◊ In Windows NT/2000/XP: TCP/IP (remember to install **omvdd.dll** file in **winnt\system32** directory)

- ❑ CA-Clipper 5.2e or 5.3 environment
- ❑ Blinker 4.0 or newer
- ❑ MEDNTX52.LIB library for Clipper 5.2e or MEDNTX53.LIB for Clipper 5.3
- ❑ MEDQB.LIB library
- ❑ SPXENV.BAT file or TCPENV.BAT file containing definitions of network parameters generated by the MEDIATOR server for Windows NT (DOS environment variables)

Before executing any application, an appropriate *XXX*ENV.BAT should be executed in order to set appropriate environment variables (do not run the script from within Norton Commander).

Harbour client

- ❑ The Windows 95/98/NT/2000/XP client should have IPX/SPX or TCP/IP protocol installed and configured. When using IPX/SPX protocol the following network protocols should be configured:
  - ◊ For Windows 95/98/Me: Microsoft IPX/SPX Compatible Transport
  - ◊ For Windows NT/2000/XP: NWLINK IPX/SPX Compatible Transport

  Please take care of the frame type compatibility on server and client.

  When using the TCP/IP protocol the following network protocol should be installed and configured on the client side:
  - ◊ For Windows 95/98/Me: Microsoft TCP/IP
  - ◊ For Windows NT/2000/XP:.TCP/IP
- ❑ The (x)Harbour environment in appropriate version should be installed (see ....\Harbour\hbreadme.txt and ....\xHarbour\hbreadme.txt files)
- ❑ The compilation and linking scripts from ...\Harbour\bin\ and ...\xHarbour\bin\ directories should be modified to match the local environment.

2. Preparation of Windows NT/2000/XP server

The Windows NT/2000/XP server should have the NWLINK IPX/SPX Compatible Transport protocol or TCP/IP protocol installed and configured. If the protocol is not installed, it can be found on installation Windows CD-ROM. The IPX configuration includes setting the internal network number (Control Panel/ Network/ Protocols/ Properties) and choosing an appropriate frame type (usually Auto Frame Type

Detection). The TCP/IP configuration includes setting the IP address, mask and gateway. Make sure the latest service pack is installed.

3. Preparation of the database server

a) Preparation of the Oracle server

The Oracle Server should be installed from the CD-ROM according to the installation manual. The database should be able to store your national characters (choose a valid Oracle character set - default is WE8ISO8859P1).

Ensuring the appropriate size of data storage

First, work out the approximate space requirements for data storage. Generally, one should reserve at least as much space in Oracle as the size of .DBF file together with indexes.

For the purpose of ensuring enough storage space, the tablespace can be extended. Some methods are available (See *„Oracle Server Administrator's Guide"*). For example, use SQL*Plus, connect with DBA privileges and execute the following SQL command :

```
ALTER TABLESPACE <tablespace_name> ADD DATAFILE
'<datafile_name>' SIZE <new_size>;
```

<tablespace_name> and *<datafile_name>* names should be replaced with appropriate names, and size should be replaced with the size of an appropriate file, for example:

```
ALTER TABLESPACE data ADD DATAFILE
'D:\ORANT\DATABASE\DATA02.ORA' SIZE 100M;
```

The command increases the *data* tablespace by an additional file D:\ORANT\DATABASE\DATA02.ORA of 100 MB size.

Changes of database parameters

It is best to set OPEN_CURSORS to value not lower than 300. This setting enables the opening and intensive work on 50 databases. It is possible to calculate necessary number by allocating a maximum of 6 cursors for one opened database.

OPEN_CURSORS parameter is set in the database initialization file with default name INITORCL.ORA.

After making all changes of initialization file, the database should be restarted.

The NLS_LANG parameter should be changed to reflect the character set chosen for the database. Up till now there is no possibility of using indexes containing national characters in the Oracle server standard edition. Only binary sorting is available,

which results in adding all national characters at the end of the list. To change NLS_LANG parameter use:

In Windows NT the *regedit* program is used for the purpose of editing HKEY_LOCAL_MACHINE/SOFTWARE/ORACLE/NLS_LANG key.

In NetWare, a regular text editor should be used: the changed line in CONFIG.ORA file (in ORANWxxx\NLM subdirectory) should be as follows:

```
NLS_LANG = AMERICAN_AMERICA.WE8ISO8859P1
```

After introducing the change restart the database.

Adding the Oracle database user accounts

Since the owners of database objects, such as tables and indexes, are the *Oracle users*, there should be at least one such a user. The following example shows how to create the user accounts with the use of SQL*Plus and SQL commands. After connecting to the database as a user with administrative privileges, (*system* for example) execute the following command:

```
CREATE USER <user_name> IDENTIFIED BY <password>
DEFAULT TABLESPACE <data_tab> TEMPORARY TABLESPACE
<temp_tab>;
```

For example, create a user test with a password test and the privilege to create objects within the *data* space, and executing auxiliary operations (such as sorting the database) in the *temp* space:

```
CREATE USER test IDENTIFIED BY test DEFAULT TABLESPACE
data TEMPORARY TABLESPACE temp;
```

After creating a user, give him the privileges to connect to the database and to create objects:

```
GRANT connect, resource to <user_name>;
```

For example,

```
GRANT connect, resource to test;
```

b) Preparing the Microsoft SQL Server

The Mediator can work with Microsoft SQL Server version 7.0 or later. The server should be installed from distribution CD-ROM according to the setup instructions. The database has to be able to store national characters, which is why during installation you need to choose appropriate code page. In order for the Mediator server to be able to store national characters, enable regional settings in Control Panel / Regional Settings. Remember that for the database server the Mediator server is the

client. Alternatively, you can turn off the character conversion during configuration of ODBC client.

Starting the server

MS SQL Server can be started automatically after booting the machine. If the other option was chosen, start the server with graphical Service Manager. All administrative tasks for the server could be executed with Enterprise Manager tool.

Ensuring appropriate storage capacity for data

Estimate capacity necessary for storing the data. Rough estimation is the same amount as .DBF together with indexes use up. Subsequently, create the database in which tables and indexes will be stored (after connecting the server choose the Action/New database option).

In order to ensure appropriate storage you can execute following actions:

- Enable automatic disk space allocation depending on actual needs (that is default behavior).
- Allocate enough space for files containing the database (you can change parameters by editing database properties).

If the database will be used for the testing purpose, set „Truncate log on checkpoint" option (Database properties / Options tab), which allows to avoid excessive growth of log files (this option is available for MS SQL Server version 7.0).

Creating database users

Create at least one database user who will be the owner of tables and indexes. The user has to have access permission to the created database (the preferable solution is to choose his default database). After creating the user (Security/Logins/New login) add the permission of creating tables, views, procedures and the like in the chosen database (database / Properties / Permissions tab).

Creating ODBC data source

Create ODBC system data source via which the Mediator server will connect to the database server. The data source is created with ODBC Administrator tool (Control Panel / ODBC Data Sources). Choose System DSN tab and press the Add button. Next, choose the SQL Server driver and enter the name of the source, description, identification method (depending on the kind of created user choose the authentication by Windows NT or by the database server). Other parameters can be left unchanged.

c) Preparing Sybase Adaptive Server Anywhere

The Mediator can work with Sybase Adaptive Server Anywhere version 6.0 or later. The server should be installed from the distribution CD-ROM according to the installation instructions.

Creating the database

Create the database that will store the tables and indexes. Use the Sybase Central / Utilities / Create Database tool. Choose location for storing data and location for storing transaction log. Installing Java is not necessary for the Mediator to work, so it can be skipped ("Install base Java classes" and "Install jConnect meta-information support"). Choose the page size (e.g. 2048 B) and national character set. After creating the database, connect to it.

The database has to feature enough storage capacity for data. Rough estimation is the same amount as .DBF with indexes together use up. After connecting to the database, open the "DB Spaces" folder and edit properties of SYSTEM space for adding appropriate number of pages (or add new DB space).

Also, change the database options of maximum number of cursors and expressions (database / Set Options). Parameters Max_cursor_count and Max_statement_count can be set to 0 (no limits) or to estimated number depending on the XBASE application. Assume about 6 cursors and SQL expressions to be necessary for each opened table.

Creating database users

Create at least one user in the database, which will be the owner of tables and indexes (Folder Users & Groups / Add User). Created user has to have permission to access the database ("User is allowed to connect" option) as well as the right to create objects ("Resource" option). Remote DBA privilege is not required.

Starting the server

Depending on the need SQL Anywhere can be started as network server (via DBSRV6) or as the local server (via DBENG6). Remember that allocating greater amount of memory improves performance of the database considerably (use option -c, e.g. -c 20M allocates 20 MB cache for reading from the database).

Creating the ODBC data source

Create the ODBC system database source via which the Mediator will connect to the database server. The data source is created with ODBC Administrator tool (Control Panel / ODBC Data Sources). Choose System DSN tab and press the Add button. Next, choose Adaptive Server Anywhere 6.0 driver and enter the name of the data source and its description. Depending on configuration complete database parameters available on the "*Database*" tab. **Turn off the "Allow multiple record fetching" option on "*Advanced*" tab.**

d) Preparing PostgreSQL

The Mediator can work with PostgreSQL server version 7.2 or later. The server should be set up according to its installation instructions.

Creating the database

First, create the database, where tables and indexes will be stored. The best solution is to use *createdb* utility.

The disk containing the database has to have enough free space available for the data and its growth. The server parameters are located by default in the postgresql.conf file. You can edit it using any text file editor. One of the changes that are necessary is turning off the full table scan feature. You can do it by adding the line: enable_seqscan=false. Other values can be changed depending on particular requirements, for example you can increase the amount of memory for shared buffers (shared_buffers parameter). You have to be careful and not exceed the available random access memory, because using virtual memory facilities can cause major slowdown.

Creating database users

In the created database you need to set up the user who will own the tables and indexes. The user can be added using SQL command (CREATE USER) or *createuser* utility (it is a script). The easiest option is to use PGAdmin GUI utility.

Creating ODBC data source

You have to create the ODBC data source on the computer where Mediator server is installed. The Mediator server will connect to the database server using this data source. In Windows NT/2000/XP environment the data source can be created using ODBC Administrator (Control Panel | ODBC Data Sources). Click tab DSN System. Click Add button. Choose the driver „PostgreSQL+(Beta)". Type in the source name and its description. In the data source configuration (Datasource | Advanced Options | Page 2) turn off LF/CR+LF conversion.

In order to use ODBC in Linux environment you need to install iODBC manager and the Postgres driver (psqlodbc). Next, configure the data source (iODBC manager libraries required by Mediator for PostgreSQL). The data source parameters are stored in file .odbc.ini in the user's home directory (as long as the location was not changed during manager installation).

Sample .odbc.ini file:

```
[ODBC Data Sources]
test          = PostgreSQL Test
```

```
[test]
Description         = Postgres SQL Test
Driver              = /usr/local/lib/psqlodbc.so
Trace               = No
Database            = testdb
Servername          = localhost
UserName            = postgres
Port                = 5432
Protocol            = 6.4
ReadOnly            = No
RowVersioning       = No
ShowSystemTables    = No
ShowOidColumn       = No
FakeOidIndex        = No
```

The file quoted above can be edited using system editor (e.g. vi) or using GUI tools installed with iODBC manager: iodbcadm-gtk. Specify the name of the data source, the driver, server name, database, correct port and the protocol.

ODBC drivers installed in the system are specified in .odbcinst.ini file (this file is not obligatory).

Sample .odbcinst.ini file:

```
[ODBC Drivers]
Postgres = Installed

[Postgres]
Driver = /usr/lib/psqlodbc.so
```

Recommended maintenance

It is very important to periodically clean the database (e.g. every night) using VACUUM utility (*vacuumdb –d dbname* or *vacuumdb –a*).

e) Preparing other DBMSs

A database server (MySQL, DB2, or other, for which Mediator server is available) should be installed according to installation instructions. Create a user, which will be able to connect to a database server and create there objects. Database files should be large enough to store all planned data. Then create an ODBC data source on the computer where Mediator server is installed. The Mediator server will connect to the database server using this data source. In Windows NT/2000/XP environment the data source can be created using ODBC Administrator (Control Panel | ODBC Data Sources). Click tab DSN System, then Add button. Choose the appropriate driver and enter data source name and description.

4. Preparing the MEDIATOR server

After installation of the server, start it according to the manual. The methods of server start-up as well as operating are described in *"Operating the Mediator server"* on page III-1.

5. Preparing the CA-Clipper client

The programmer's workstation does not need much preparation. The DOS system should be installed together with Netware clients, CA-Clipper 5.2e or 5.3, Blinker 4.0+ and the MEDIATOR client. Before starting application, the following DOS environment variables should be set: MEDNETADDR, MEDNODEADDR and MEDSOCKET (SPXENV.BAT file or TCPENV.BAT depending on network protocol). If TCPENV.BAT generated by the MEDIATOR server is used, it is necessary to complete the MEDNODEADDR parameter (SPXENV.BAT file does not require to be completed). Sample programs and a compilation script are located in the SOURCE\UTIL directory.

DOS environment variables

In order to connect to the MEDIATOR server, CA-Clipper application should identify a server address. The server address consists of three elements, which are read by an application from the appropriate DOS environment variables or MEDAPP.INI file.

These are:

MEDNETADDR – network address

MEDNODEADDR – node address

MEDSOCKET – network port

The above parameters can be found automatically by the MEDIATOR server after starting it on the Windows NT/2000/XP server, and saved in the file named by the user.

For the communication over SPX:

MEDNETADDR – a hexadecimal internal network address – on Windows NT/2000/XP can be read using **Settings/ControlPanel/Network/Protocols/NWLink IPX/SPX Compatible Transport/Properties/General/Internal Network Number**. If the internal network on Windows NT/2000/XP is not defined (it holds 0 value), the unique number of a given network environment should be set. For example, if internal network number 123 is configured on the server :

then the MEDNETADDR parameter should be set to:

```
SET MEDNETADDR=00000123
```

(it is necessary to complete the parameter with zeros up to 8 digits)

MEDNODEADDR– if MEDNETADDR is set to the address of the server internal network, the node address should be set to 1:

```
SET MEDNODEADDR=000000000001
```

(it is necessary to complete the parameter with zeros up to 12 digits)

MEDSOCKET – a default SPX port for the MEDIATOR server is 4546:

```
SET MEDSOCKET=4546
```

The contents of SPXENV.BAT file for the internal network 123 can be as follows:

```
SET MEDNETADDR=00000123
SET MEDNODEADDR=000000000001
SET MEDSOCKET=4546
```

For the communication over TCP:

MEDNETADDR – this value is unused and can be ignored

MEDNODEADDR – the server IP address should be set (for the Windows NT/2000/XP server a parameter can be taken from the TCP/IP configuration: **Settings/Control Panel/Network/Protocols/TCP IP**. For example:

```
SET MEDNODEADDR=10.19.1.1
```

MEDSOCKET – a default TCP port for the MEDIATOR server is 19C8

```
SET MEDSOCKET=19C8
```

The contents of TCPENV.BAT file for the server address 10.19.1.1 can be as follows:

```
SET MEDNETADDR=
SET MEDNODEADDR=10.19.1.1
SET MEDSOCKET=19C8
```

The database login parameters (automatic logging in)

If *noautlog.obj* file is linked into a program, the application itself is responsible for connecting to the Mediator server by calling the *MedLogin()* function. Otherwise, there will be an attempt to connect to the Mediator server immediately after starting application.

Application will attempt to find values of the following parameters

MEDNETADDR
MEDNODEADDR
MEDCS
MEDSOCKET
MEDUSER
MEDPASSWD

Following steps will be executed to get parameters values

1. Application will attempt to read value of MEDINIFILE environment variable which can specify alternative name of the file containing configuration parameters. If variable is not set, application will assume parameters are defined in MEDAPP.INI file located in current directory.

2. Application will attempt to read value of MEDCONN environment variable which can specify section of the configuration parameters file, where connection parameters are defined. If MEDCONN is not defined, application will assume parameters are defined in root section (the ones with no section specified).

3. For each parameter independently application will

- try to read its value from the environment using GETENV() function
- if not successful, application will try to read parameter value from configuration file, from section calculated in 2. step

If MEDNODEADDR, MEDUSER and MEDPASSWD parameters are calculated during previous steps, application makes an attempt to connect to Mediator server using calculated values. If values for parameters MEDSOCKET and MEDCS are not known, values 19C8 and "" (empty string) respectively will be used when connecting.

In case not all three mentioned parameters are calculated, application will display the form to allow you enter required parameters (user name, password, database, server IP address and server port number).

Sample MEDAPP.INI file containing connection parameters is located in BIN subdirectory of Mediator client.

Attention!

MEDCS – database; for Oracle it is a name of a service recognized by SQL*Net on server (if it is not given, the connection is directed by default via LOCAL parameter set on the server). For other servers it is ODBC data source name on the computer when Mediator server is installed.

Example:

```
SET MEDUSER=test
SET MEDPASSWD=test
SET MEDCS=test_ipc
```

WARNING!

While executing the CA-Clipper/MEDIATOR application in Windows 95/98/Me or Windows NT/2000/XP environment make sure to turn **off** (uncheck) an option of suspending the operation in the background (**Properties/Misc/Background**: Always suspend).

6. Setting up the (x)Harbour client

The Mediator client for Harbour can be installed on the following platforms: Windows 95/98/Me/2000/NT. It is recommended to install the client software using the provided installation program. The Harbour client can be installed in the same directory as the clients for Clipper and/or CA-VO. After installation of the client software, it is recommended to add the name of the directory containing binaries (....\Harbour\Bin and ....\xHarbour\Bin) to the system path. The compilation and linking scripts: from ...\Harbour\Bin and ...\xHarbour\Bin directories should be modified to match the local (x)Harbour compiler installation by writing the appropriate directory names in the definitions of variables.

IMPORTANT!

At the moment of this writing the following Mediator client libraries for (x)Harbour in Windows environment are available:

- …\b32 - compiled with Borland C++ 5.5 – compatible with official (x)Harbour build compiled with Borland compiler
- …\vc – compiled with MS VC++ 6.0 SP5 with /TP switch – compatible with official (x)Harbour build compiled with MSVC compiler
- …\xcc - compiled with MS VC++ 6.0 SP5 without /TP switch – compatible with commercial xHarbour distribution available from xHarbour.com Inc. (www.xharbour.com)

Remember that the version of (x)Harbour should be compatible with the version of Mediator libraries. The version of the Harbour or xHarbour software which can be used with the provided Mediator software is always mentioned in the ....\Harbour\hbreadme.txt and ....\xHarbour\hbreadme.txt files installed together with the client software.

Establishing the connection between the (x)Harbour application and the Mediator server

For the Harbour application to co-operate with the Mediator a connection between the application and the server should be established. Connection is established in exactly the same way as for CA-Clipper application. In automatic login mode, connection parameters are read from application environment. If not present in environment, they are read from MEDAPP.INI configuration file.

The (x)Harbour Environment

When developing the Harbour application using the Mediator client software you need to add MEDNTX.LIB library to the current project. This library contains both MEDNTX and MEDCDX drivers. Include the following command in one of the application files:

```
REQUEST MEDNTX
```

no matter which driver is actually used.

Sample compilation and linking scripts distributed together with the client software do link MEDNTX.LIB library.

If the developed application uses Mediator extensions, sometimes it might be necessary to include the Mediator header file using the statement:

```
#include "mediator.ch"
```

in each application source file (.PRG) using the extensions.

Debugging in the (x)Harbour environment

It is possible to debug the (x)Harbour applications linked with the Mediator driver. While debugging the application the connection to the Mediator server can time out (typically after 1 min. of idle time on the connection link). To avoid this add the following line at the beginning of the program, after establishing the connection to the Mediator server:

MedDiscTm(65535)

Calling this function will disable standard time-out disconnection procedure and will allow to effectively debug the application. After the debugging is finished comment out (or delete) the mentioned line.

7. Compatibility between the versions of the Mediator client and the server

The Mediator client checks compatibility between the client and server versions while connecting to the server. The versions are designated by four numbers, for example: 1.4.0.2, where subsequent numbers designate: the main version number (1), the detailed version number (4), the patch level (0), and the small patch level (2). The client connects to the server, if the differences between the number of version exist in the last position (i.e. in the small patch number). If those differences exist in any other position, the client will refuse to connect, for example:

The client version number 1.4.0.2 Server version number 1.4.0.5 – a connection will be established.

Client version number 1.4.1.5 the server version number 1.4.0.5 - a connection will not be established (because of the difference in the patch number).

III. Operating the Mediator server

Some of Mediator servers (depending on the platform) are equipped with the user interface limited to *On/Off* function and view of base operating and licensing informations (Windows NT/200x/XP Mediator desktop version, NetWare). Mediator servers run as WinNT services or UNIX releases have no any user interface. All the servers, independently on the platform, can be configured and managed via network using "MMT" application (*Mediator Management Tool*) which can be run on any Windows machine. The application is distributed together with Mediator. MMT is described in section "*An external configuration program with a monitor for Mediator servers*" on page III-7.

1. Mediator for Windows NT/2k/XP (desktop version)

a) Starting the server

The Mediator server for Windows NT is started just like any other Windows application – immediately by an application icon, via the shortcut to the application or from a command line. The server is not ready immediately after start and it does not accept connections from the client stations. In order to activate the server, use the Start button on the main panel. You can start the server directly in the mode of waiting for a connection. In such a case, an **/s** or **–s** argument should be added to the command line. It is only possible when starting the program from the command line or by the shortcut, where in *Properties* **/s** or **–s** argument can be added to the program execution line (*Target*).

```
mediator /s
```

or

```
mediator –s
```

b) Configuration and management

Adjusting software options, user accounts manipulation and server monitoring are possible only via network using MMT application described in chapter "*An external configuration program with a monitor for Mediator servers*" on page III-7.

c) The main panel of the Mediator server for Windows

After starting the Mediator server for Window, a panel which controls the main functions of the server and displays basic information about the server state appears.

Program name together with its version are displayed in the window title bar (the first two version digits). The exact version number (consisting of four numbers) is also displayed in *About* dialog window, accessible from the main panel. The paragraph below describes the meaning of particular items of the panel.

- The graphical semaphore shows the current state of the server (depending on options, it is displayed in the black and white or color mode):
  - Red light and *Resting...* are displayed. The server is in the resting mode; processes that are waiting for the client connections are not working,
  - Yellow light and *Starting...* are displayed. The server is in the phase of startup and configuration of processes that are waiting for connections,
  - Yellow light and *Stopping...* are displayed. The server is in the phase of disconnecting sessions and terminating working threads,
  - Green light and *Running...* are displayed. The server is working; connected clients are served and the server is awaiting connections from new clients.

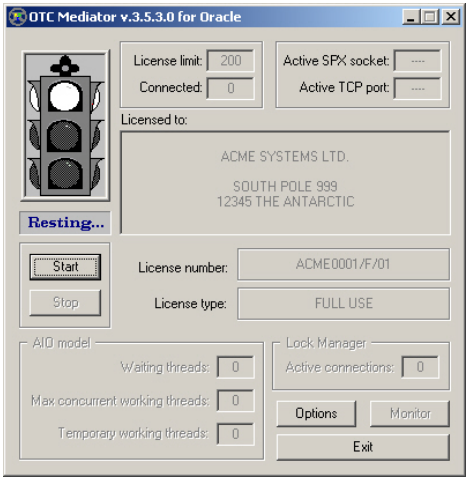


Figure 4. The main panel of Mediator server for Windows in the non-active mode

- The *Start* button starts the server, i.e. provokes the transition from the resting mode to waiting for new client connect requests. During the operation of the server, the *Start* button is not active.

- The *Stop* button makes the server switch to the resting mode. If users are connected to the server, then after the confirmation of request to close the server down, their sessions are closed. The button does not cause exiting the server application. In the resting mode, the *Stop* button is not active.
- The *Users* frame specifies the possible maximum number of concurrent sessions (*License limit* field), and the number of current connections from clients (*Connected* field).
- The network protocol frame. If the server is in the resting mode, no information is displayed in fields. During the operation, the *Active SPX Socket* field displays the number of SPX socket on which the server awaits connection requests from clients using the IPX/SPX protocol, while *Active TCP Port* field displays the number of TCP port, on which the server awaits connections from workstations using the TCP/IP protocol. In case of conflicts with other network applications working on the server, the default TCP port values or SPX socket can be changed in the Mediator configurator.
- The window *Licensed to* contains the name and address of the Mediator server end-user. The producer (OTC) sets the name.
- The frame *License number* contains the Mediator server license number granted for the user, whose name is displayed in the window *Licensed to*, or the word TEMPORARY for the demo and evaluation servers.
- The frame *License type* designates the type of license granted to the user:
  - FULL USE – a full commercial license,
  - FOR NON-COMMERCIAL USE – a license limited to development, testing and demonstrations of the Mediator-based application software
  - DEVELOPMENT – exclusively for the development, testing and demonstration of the application software based on the Mediator package
  - EVALUATION – a limited period license for the purpose of testing the application software at the customer's place; the frame displays the termination date of the evaluation period as well.
- The frame *AIO model* specifies Mediator state while working in AIO mode (only for WinNT/2k/XP versions). If the server does not work in AIO mode the frame is inactive.
- The frame *Lock Manager* displays Lock Manager connections states. If the server does not cooperating with Lock Manager, the frame is inactive.
- The *Exit* button closes server application. It is active only during the resting mode of the Mediator server.

2. Mediator for Windows NT/2k/XP (service version)

a) Starting the server

After the server (**medsvc.exe)** is installed it starts automatically when Windows starts. This version of Mediator does not require WinNT/2k system login nor any additional operations. In order to start or stop the service use applet Services from Windows Control Panel (see Windows NT or 2k manual).

b) Configuration and management

Adjusting software options, user accounts manipulation and server monitoring are possible only via network using MMT application described in chapter "*An external configuration program with a monitor for Mediator servers*" on page III-7.

3. Mediator for NetWare

a) Starting the server

The Mediator server for NetWare is started with the NetWare *load* command. The server is not active immediately after the start and it does not await connections from the client workstations. In order to activate the server, use Alt+F2 keys on the main panel. It is possible to start the server directly in the mode of waiting for connections. For that purpose, add the **/s** or **–s** argument to the command line.

```
load mediator /s
```

or

```
load mediator –s
```

b) Configuration and management

Excepting listening port and socket number replacing, all the operations such as adjusting software options, user accounts manipulation and server monitoring are possible only via network using MMT application described in chapter "*An external configuration program with a monitor for Mediator servers*" on page III-7.

c) The main panel of the Mediator server for NetWare

After starting the Mediator server for NetWare, the main functions control and information display panel appear.

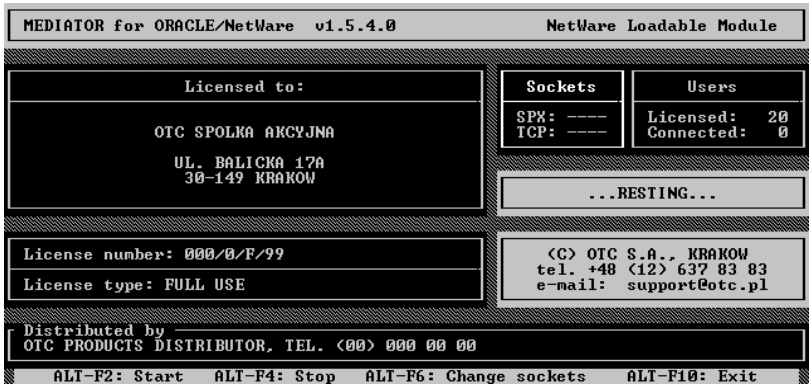


Figure 5. The main panel of the Mediator server for NetWare in the non-active mode

The name with the full software version number is displayed at the top of the window. Descriptions of particular elements of the panel are given below.

- The *Licensed to* frame contains the name and the address of the end user of the Mediator server. The name is entered directly by the software manufacturer (OTC).
- The *License number* frame contains the license number of the Mediator server granted to the user, whose name is displayed in the *Licensed to* frame or the TEMPORARY string for demo and evaluation servers.
- The *License type* frame designates the type of the license granted to the user:
  - FULL USE – the full commercial license
  - DEVELOPMENT – the license granted exclusively for the purpose of development, testing and demonstration of application software based on the Mediator server
  - EVALUATION – the limited time license granted for the purpose of testing of the application software on the client's site (the frame displays the expiration date of the evaluation period as well).
- The *Distributed by* frame contains the name of the Mediator server distributor. The name is entered directly by the software manufacturer (OTC).
- The *Sockets* frame – when the server is in the resting mode, windows do not display any information. During operation, the *SPX* field contains the number of socket, on which the server awaits connection requests from clients using the IPX/SPX protocol, while the *TCP* field displays the TCP port number on which the server awaits connection requests from clients using the TCP/IP protocol. In case of conflict with other network applications operating on the server, default TCP or SPX socket values can be changed with Alt+F6 keys.

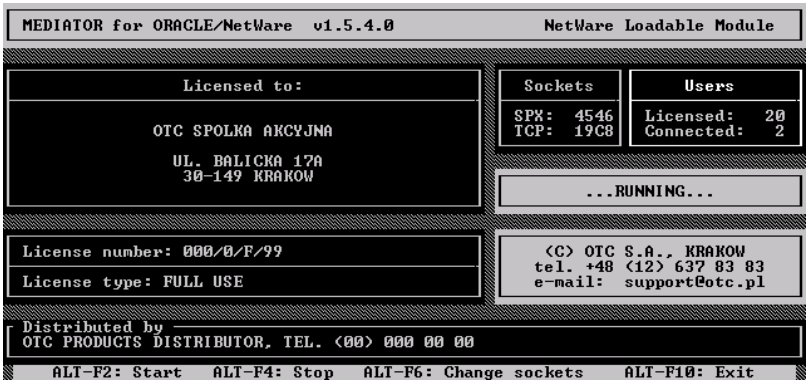


Figure 6. The main panel of the Mediator server for NetWare in the active mode

The *Users* frame displays the maximum number of concurrent sessions allowed (the *Licensed* frame), and the number of currently connected clients (*Connected* frame).

- The white field below the *Users* and *Sockets* frame indicates the current state of the server:
  - ...*RESTING*... – the server is in the resting mode; processes waiting for connection requests from clients are not running;
  - ...*STARTING*... – the server is in the phase of startup and configuration of the listening processes;
  - ...*STOPPING*... – the server is in the phase of disconnecting all possible sessions and closing working threads;
  - ...*RUNNING*... – the server is functional; it is serving connected clients and awaiting connection requests from new clients.
- The white field contains information about the software manufacturer (OTC).
- Pressing Alt+F2 keys switches the server from the resting mode to the mode of waiting for connection requests from clients.
- Pressing Alt+F4 keys switches the server to the resting mode; if users are connected to the server, then after confirming the request for closing the server, their sessions are disconnected. The button does not close the server completely.
- Pressing Alt+F6 keys starts the edition of the SPX socket number as well as the TCP port number. The change saves new values automatically to the configuration file.
- Pressing Alt+F10 keys closes the server completely. If the server is in the active mode, then it is first switched to the resting mode. If users are connected to the server, then after confirming the request for closing the server, their sessions are disconnected.

4. An external configuration program with a monitor for Mediator servers

All supported Mediator servers can be configured using **Mediator Management Tool (mmt.exe)** working in Microsoft Windows XP/2003/7/Vista/2008 environment. MMT communicates with Mediator using IP or IPX protocol excepting UNIX platforms where only IP protocol is available for communication. MMT can manage multiple Mediator servers; however, they all have to be of the same version as MMT. The program console allows to define up to eight servers and maintain simultaneous connections with them.

a) MMT application console

Figure 7 shows the main MMT application window. It contains the server list and a number of buttons that allow for various actions related to the list and the application.

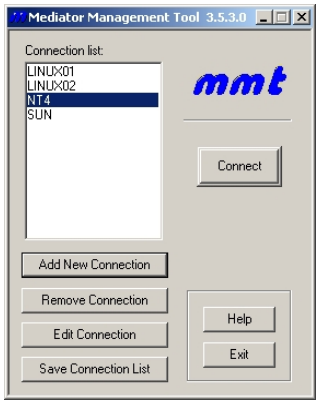
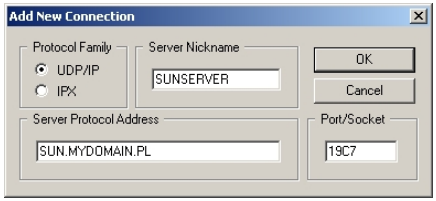


Figure 7. The main panel of MMT

Defining, deleting and editing connections

After the first MMT start-up, the list of connections to managed servers is empty. You should use the buttons on the console to create, edit and delete connections. The short description of the buttons follows:

- The *Add New Connection* button opens up the dialog for creating new connections.



In order to add a connection, fill in the following fields:

- *Protocol Family* – communication protocol (default is UDP/IP),
- *Server Nickname* – a connection name (alias), which will identify a given server in all the MMT dialogs,
- *Server Protocol Address* – the address of a server where Mediator is installed – if a connection uses IP address, the address can be specified in a number format *xxx.xxx.xxx.xxx* (e.g. 100.110.120.130) or by the usage of a domain name (e.g. alfa.acme.com). In the case of IPX protocol, the address should to be specified in the following format: *net_number:node_number* (e.g. 01234567:0123456789AB),
- *Port/Socket* - UDP/IP port number or IPX socket number (in hexadecimal notation, e.g. 19C7) – the default UDP/IP port, where Mediator listens to MMT commands, is 19C7hex; the default IPX socket number for the communication with MMT server is 4545hex.

After clicking *OK* button the correctness of the entered data is verified and, as long as the address was specified as a host.domain name, it will be tried to be changed into a numerical format. If all the data is correct, a connection name will appear in the main dialog. If any field of the *Add New Connection* dialog has been filled incorrectly, the program will display an appropriate message.

After defining the maximum number of connections, the *Add New Connection* button becomes inactive. It can be used again after deleting at least one connection from the list.

- The *Remove Connection* button is used for deleting connections from the list. To delete a connection, select a server name on the list and click *Remove Connection*. After confirming the action, the connection will be removed from the list. If there are no defined connections on the list, *Remove Connection* button is inactive.
- The *Edit Connection* button is used for changing parameters of a defined connection. After clicking on this button, a dialog similar to the one that allows to add connections is opened.

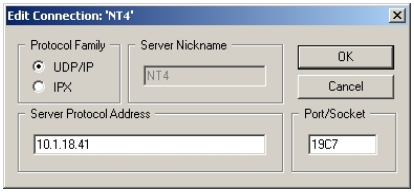


Figure 9. Dialog „*Edit Connection*"

That dialog allows to edit all the parameters of a connection, except from the server name (*Server Nickname*). The meaning of particular fields is the same like in case of adding a new connection.
After clicking *OK* button the correctness of the entered data is verified and, as long as the address was specified as a host.domain name, it will be tried to be changed into a numerical format. If all the data is correct, a connection name will appear in the main dialog. If any field of the *Edit Connection* dialog has been filled incorrectly, the program will display an appropriate message.
If the list does not contain any defined connections, the *Edit Connection* button is inactive.

- The *Save Connection List* button saves parameters of defined connections into the system registry, in order for them to be stored after closing the MMT application.

Connecting to Mediator server

In order to connect to a selected server, click its name on the list and use *Connect* button. If the server is available, a connection will be established and the server will send back the basic information regarding it, and a new window, titled *Description of 'NAME'* will be displayed. This window is described in section "*Server description (window: Description of...)*" on page III-10. You can simultaneously connect to several servers. If a required server is unavailable, an appropriate message will be displayed.

Help

After clicking the button *Help* located on the console, a document that contains a summary of basic MMT features will be displayed.

Exiting program

Use the *Exit* button to terminate MMT program. After clicking this button, all the windows of an application will be closed. If a connection list has undergone any changes, the program will suggest saving those changes, unless a user has already done that himself using *Save Connection List* button.

b) Server description (window: *Description of...*)

After connecting with the managed server, a window titled *Description of 'connection_name'* is displayed. This window contains the following information:

- *OS* – the operating system of the managed Mediator.
- *Database* – a type of a database that Mediator communicates with (Oracle, MS Sql Server, Sybase Adaptive Server Anywhere, PostgreSQL).
- *SPX socket* – IPX/SPX socket number, where Mediator awaits for requests from clients.
- *TCP port* – TCP/IP port number where Mediator awaits requests from the clients.
- *Active from ...* – the time which passed from the last start-up of the server (the format is *days/hours/minutes/seconds*).
- *Licensed for* – the maximum number of simultaneous session (*apps*) or devices (*devs*) that can be connected simultaneously to a given server (according to the licence granted for a server),
- *Connected apps* – the number of applications connected to the server at the moment,
- *Connected devices* – the number of devices (computers) connected to the server at the moment,
- *Licensed to* – the name of Mediator license owner,
- *Lic. number* – the serial number of license granted for using a server (or TEMPORARY for evaluation and demo servers),
- *Lic. type* – the type of a particular license: full, a developer license, runtime, evaluation with the expiry date.

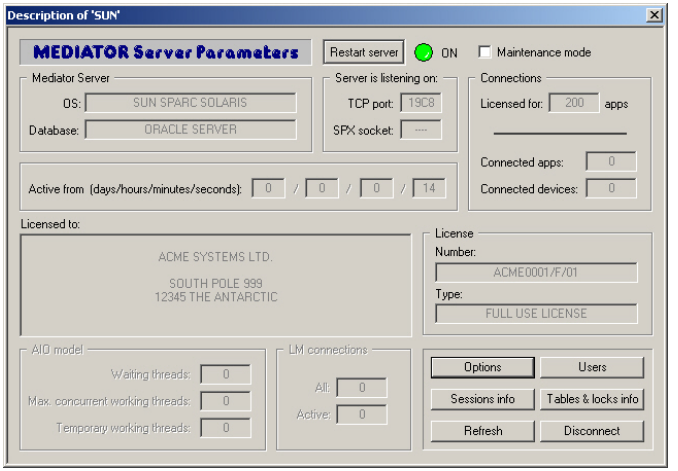


Figure 10. Dialog „*Description of...*"

Additionally, the window displays two frames containing information fields regarding an asynchronous mode of operation for a server (*AIO model* – only Windows servers) and for the mode of operation involving Lock Manager:

- *AIO model - Waiting threads* – this item specifies the number of threads awaiting assignment for a task execution.
- *AIO model - Max concurrent working threads* – the maximum number of threads simultaneously executing tasks (since the server start-up until now).
- *AIO model - Temporary working threads* – temporary number of threads simultaneously executing tasks (measured throughout the past few seconds).
- *LM connections - All* – the total number of all connections to the Lock Manager.
- *LM connections - Active* – the total number of active connections to the Lock Manager.

Apart from information fields, the window contains a number of buttons for a configuration of a connected server:

- *Restart server* – it restarts the managed server, causing a disconnection (*abort*) of all users.
- *Maintenance mode* – the managed server is automatically switched to *Maintenance* mode, in which it does not accept any connections from users. A restart in the maintenance mode disconnects all the users and allows for necessary maintenance and configuration of Mediator software and a database.
- *Options* – this item opens a dialog titled *Configure Server 'NAME'*, which allows for a modification of server parameters. An extended description of this dialog is available in chapter „*Editing Mediator server parameters (dialog: Configure Server...)*", page III-12.
- *Users* – this item opens a dialog titled *Users of 'NAME,* which allows for manipulating user accounts of Mediator server. An extended description of this dialog is available in chapter "*Defining, editing and deleting users of the Mediator server (dialog: Users of...)*" on page III-15.
- *Sessions info* – this item opens a system monitor window informing about sessions and users connected to Mediator server and the resources engaged.
- *Tables & locks info* – opens up a system monitor window informing about opened tables and writing locks on records or tables.

Remaining buttons:

- *Refresh* – this button refreshes information displayed in the window *Description of...* by sending a new query to the connected server.
- *Disconnect* – this button disconnects Mediator; the windows and dialogs related to that connection are closed.

c) Editing Mediator server parameters (dialog: *Configure Server...*)

Opening *Configure Server...* dialog causes that current parameters are read from the server. The parameters are displayed in dialog fields and can be changed.

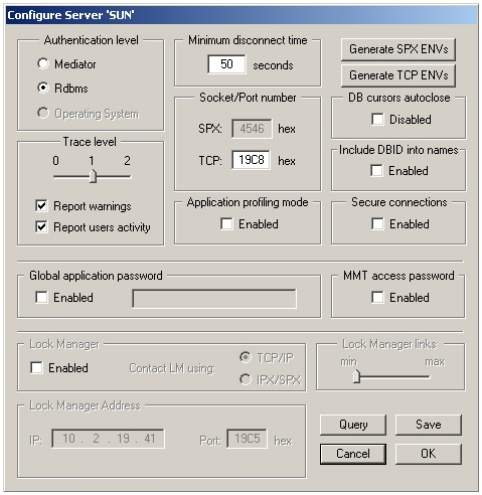


Figure 11. Dialog „*Configure Server…" (no Lock Manager)*

Below there is a list of configurable Mediator server parameters together with short descriptions:

- *Authentication level* defines a method of added users' identification. If it is set to **Rdbms**, then a username and a password received from the application are sent directly to the database server. When the parameter is set to **Mediator,** a username and a password are checked in Mediator server; but Rdbms username and password, assigned to a given Mediator user is sent to the database server. The assignment can be made during defining Mediator and Rdbms users. The third level of authorization is **OS** (*Operating System*), where users are identified by the operating system. In Mediator server OS usernames which we want to use must be defined, together with the appropriate assignments to specific Rdbms users.

  A user authorization on the level of Mediator server is very useful when the system administrator wants to hide a password and a username, where the database is stored, from other users of the system. The OS level authorization is equally useful, as it also allows for managing user passwords on the operating system level. This feature is implemented only on selected platforms. In case

when Mediator does not support the OS level authorization, the checkbox in the configuration program is inactive.

- *Trace level* – it specifies the number of messages stored in logs by Mediator server. Level 0: minimum number of messages – only the most important ones, affecting current system operation; level 1: normal – the current server activity is logged, level 3: this level should be turned on only for troubleshooting or problem diagnostics; the number of messages stored is high.
- *Report warnings* – this options turns Mediator server activity reporting on or off.
- *Report users activity* – this option turns on/ off the reporting of users log ins and outs.
- *Min. disconnect time* is a parameter specifying what is the minimum time (30 seconds minimum) that passes by until the server recognizes the workstation as turned off, and subsequently frees all resources associated with the workstation. This parameter can be also changed individually for each session by sending an appropriate command from a running application.
- *Socket/Port number* specifies:
  - SPX socket number, where the server awaits requests from a workstation using IPX/SPX protocol.
  - TCP port number where the server awaits requests from a workstation using TCP/IP protocol.

  In case of a conflict with another network applications installed on the server, you can change default values for the following: SPX socket = 4546 hex, TCP port = 19C8 hex.
  The new TCP port or SPX socket value is effective only after the next Mediator server start-up. If one of network protocols is not recognized on the server, a respective dialog field is inactive.
- *Async IO server model* – it turns on the asynchronous mode of operation for Mediator server (it limits the number of working threads assigned for executing particular application commands).
- *DB cursors autoclose* – this option automatically turns off the closing of unused cursors in the database (the default is on).
- Buttons *Generate SPX ENVs* and *Generate TCP ENVs* create configuration scripts for workstations. The scripts set up the environment variables that allow a workstation find Mediator server in a local or a wide area network. The values placed in scripts by Mediator can also be used for making correct settings in Windows registry during the configuration of CA-VO client. Pressing the button will display a dialog offering a saving of a file on a disk (by default the file named respectively `spxenv.bat` or `tcpenv.bat` is saved in a current directory).

Example (file SPXENV.BAT)

```
SET MEDNETADDR=0000000A
SET MEDNODEADDR=00ABCDEF0123
SET MEDSOCKET=4546
SET MEDUSER=
SET MEDPASSWD=
SET MEDCS=
```

Example (file TCPENV.BAT)

```
SET MEDNODEADDR=0.0.0.0
SET MEDSOCKET=19C8
SET MEDUSER=
SET MEDPASSWD=
SET MEDCS=
```

Due to the properties of TCP/IP protocol, the field MEDNODEADDR can not be automatically filled with a correct IP address. A user should replace 0.0.0.0 entry with a correct server address. Environment variables with no value are not obligatory.

- *Global application password* – if this option is set, each user that types in a password to his account, additionally has to add a global access password to Mediator server (a password should be typed in square brackets after the private password).
- *MMT access password* – when this option is set, an operator connecting to a server via MMT has to enter the password.

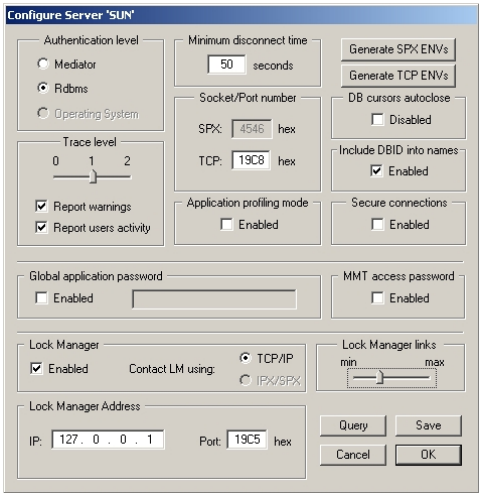


Figure 12. Dialog „*Configure Server…" (server operation involves using Lock Manager over IP protocol)*

- *Lock Manager - Enabled* – if this option is set, an operator connecting to a server via MMT has to type in the password.
- *Contact LM using* – selecting the transmission protocol to Lock Manager (TCP/IP or IPX/SPX). If the protocol is not available on Mediator server, the button is inactive.
- *Lock Manager links* – this option increases the number of connections to Lock Manager. The number of connections is calculated using the number of users and it is weighted using the parameter *Lock Manager links*.
- *Lock Manager Address* – IP or IPX address (hexadecimal) and port/socket used for a communication with Lock Manager.

- The *OK* Button closes the configuration window. If a user has made any changes and they have not been confirmed, the program will offer saving the changes in the configuration file on the server.
- Button *Save* confirms changes of parameters (i.e. it sends them to Mediator and stores on the server disk).
- Button *Query* allows to re-read parameters from the server.
- Button *Cancel* closes the configuration window. Changes made configuration parameters will stay locally in MMT structures until the contents are refreshed by using *Query* button or the program is terminated.

d) Defining, editing and deleting users of the Mediator server (dialog: *Users of...*)

Dialog *Users of...* allows to manage users' accounts of Mediator server. The only purpose of those accounts is to identify users in the authorization mode on the level of Mediator or the operating system (see *Authentication level*). If a system administrator chooses an authorization to be executed via RDBMS, the account data will not be used.

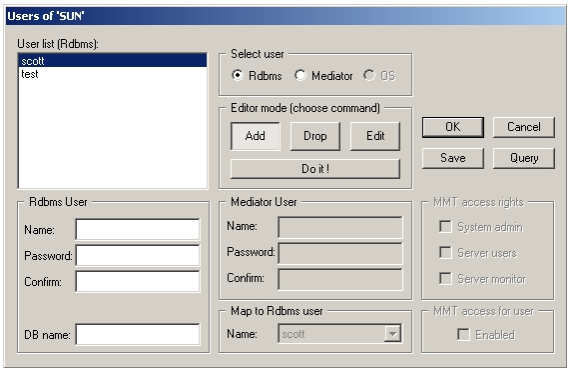


Figure 13. Dialog „*Users of...*" in user adding mode

This dialog allows to add, edit and delete accounts. This can be done on various levels: Mediator, operating system and RDBMS as well. RDBMS users' data serves only to define the assignments between Mediator and RDBMS accounts or/and between the operating system and RDBMS accounts.

Opening of the dialog *Users of...* causes automatic reading of the information about users and the assignments between Mediator/OS/RDBMS server accounts. If there are no defined accounts on the server, an appropriate message will be displayed before opening the dialog. The main item of the dialog is a list of users (*User list*) where names of defined users accounts are displayed. The type of displayed accounts (Mediator/RDBMS/OS) can be changed by using *Select user* button. If the server does not support operating system accounts, the button is inactive. Together with an account type change, some fields, crucial for entering data about users will activate; and unnecessary fields will become inactive. The dialog operates in three modes: adding, deleting and editing users. Descriptions of actions necessary to execute those operations will follow below.

Adding user accounts

In order to add a user account you need to switch the dialog into an adding mode using the button *Add.* Next, depending on a type of a user, enter relevant information into active fields and apply the changes using button ***Do it!***:

1. RDBMS user:
   - *Name* – an RDBMS user name.
   - *Password* – an RDBMS user password.
   - *Confirm* – type the password again in order to verify it.
   - *DB name* – a database name (*alias/connect string*), where the account is defined.

   WARNING!

   Please remember that creating RDBMS account in Mediator does NOT result in creating a relevant account in the database server!

   WARNING!

   The verification of the entered RDBMS user is NOT executed on the database server. Therefore, it is possible to enter a non-existent account. The configuration program only checks whether all the necessary fields have been filled in and if the content of a password field agrees with the content of a verification field.

2. A Mediator user:

- *Name* – a Mediator user name.
- *Password* – a Mediator user password.
- *Confirm* – type the password again in order to verify it.
- *Map to RDBMS user* – a drop-down list, containing RDBMS accounts – choosing one of these accounts means that every user connecting to a defined Mediator account, actually connects to the database account defined in the field *Map to RDBMS user*.

☝ **WARNING!**

If there is no RDBMS account existing during entering a Mediator user data, the configuration program will not allow to save the new user (a relevant message informing about that event will be displayed). That is why, the first entered user should be RDBMS one.

3. An operating system user:

- *Name* – an account name,
- *Map to RDBMS user* – a drop-down list, containing RDBMS accounts – choosing one of these accounts means that every user connecting to a defined OS account, actually connects to the database account defined in the field *Map to RDBMS user*.

☝ **WARNING!**

The verification of an entered RDBMS user is NOT executed on the database server. Therefore, it is possible to enter a non-existent account.

☝ **WARNING!**

If there is no RDBMS account existing during entering the operating system user data, the configuration program will not allow to save the new user (a relevant message informing about that event will be displayed). That is why the first entered user should be RDBMS one.

After accepting the data, a user name will be displayed on the list.

Deleting user accounts

In order to delete an account, switch the dialog to the deleting mode by clicking the *Drop* button and select the user type (RDBMS/Mediator/OS) by using the *Select user* button. Next, select the user account on the list by using the mouse and confirm the action by clicking ***Do it !*** The user name will be deleted from the list.

☝ **WARNING!**

Deleting RDBMS user will not succeed, if s/he is assigned to at least one RDD or OS user.

Editing user accounts

In order to edit an account, switch the dialog to the editing mode by clicking *Edit* button and select the user type (RDBMS/Mediator/OS) by using the *Select user* button. Next, using the mouse , select the user account on the list to be modified and confirm the action by clicking ***Do it !***You can change all the data except from an account name.

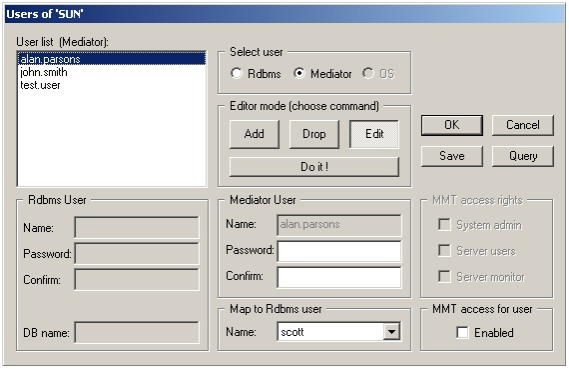


Figure 14. Dialog „*Users of...*" in user data editing mode

MMT access rights

After the *MMT access password/Enabled* checkbox in the *Configure* form is checked, every connection between MMT and Mediator server will require authorization with the username and the password. MMT access rights can be granted to each '*Mediator user'* account together with database access permissions. It is enough to check the *MMT access for user/Enabled* checkbox during account creating or editing and then grant required role by checking one or some of *MMT access rights* checkboxes. There is also a possibility of creating accounts with MMT access only (without database permissions). In such a case it is neccessary to choose *~MMT only~* entry from *Map to Rdbms user/Name* listbox.

MMT user roles

System admin – all MMT functions including creating/editing MMT and database users are permitted.

Server users – only account creating/editing functions are permitted.

Server monitor – only system monitor related fuctions are permitted.

e) System monitor

Mediator server monitor provides a system administrator with information about connected users, opened tables and table locks. The monitor is divided into two parts: a user session monitor and a resource monitor (tables, locks). The session monitor is started in the server description window (*Description of...*) by using the *Sessions info* button and the resource monitor is started by clicking *Tables & locks info*.

Resource monitor

A resource monitor allows to view opened tables, as well as tables and record locks in Mediator server. The resource monitor consists of several dialogs:

The list of opened tables and record locks (FLOCKs)

The dialog „*Tables & locks on...*" consists of two parts.

The main part (the large list) contains the information about all the opened tables in the system. One row on the list describes one opened table and contains:

- *Name* – the name of a table as seen by the application (Clipper).
- *Rdbms name* – a table name on the database server,
- *Owner* – the table owner (as defined by a database server account),
- *Mode* – a table open mode (*SH* – shared, *EX* – exclusive),
- *LM* – a local table (*LOC*), a global table (*GLB* – when synchronised by Lock Manager) or by an unknown type (*UNK*),
- *Users* – the number of users who have this table opened,
- *Flock* – meaning a blocked table - if not, the value is *NO*. If it is locked, the value is ID of Mediator server session that has locked a given table,
- *Columns* – the number of columns in a table,
- *Indexes* – the number of indexes in a table.

The second part of the dialog (the smaller list), titled *Locked tables (FLOCKs)*, displays information about all the tables locked by the function FLOCK(). Every row describes one opened table and contains:

- *Name* – a table name,
- *Owner* – a table owner (as defined by a database server account),
- *Locked by* – ID of the session that has locked this table.

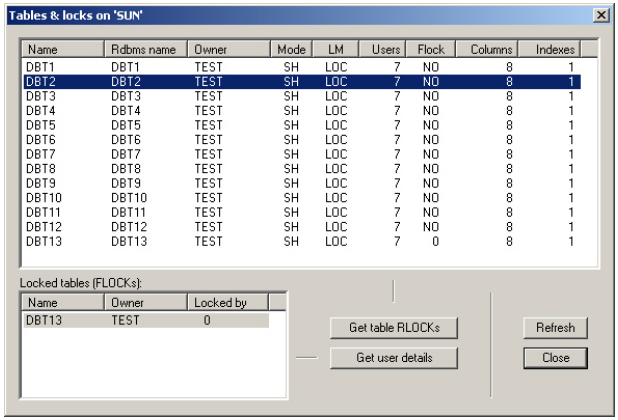


Figure 15. Dialog „*Tables & locks on...*"

Buttons:

- *Get user details* – after selecting one of the tables from list of locks, the window with a description of the session that has locked this table (fig. 10) is opened. If there are no tables present on the list of locks, the button is inactive.
- *Get table RLOCKs* – after selecting a table from one of the lists, the window containing the list of locked records in this table (fig. 11) is opened .The hyphen located over or by the side of the button indicates the list where the table has been selected from.
- *Refresh* – refreshes the window contents.
- *Close* closes the dialog.

Details of the session locking a table

Clicking *Get user details* in the window *Tables & locks* displays the dialog „*User details from...*". It displays basic information about a user or a session locking a particular table.

- *Rdd connection ID* – the session ID, assigned by Mediator server during logging in,
- *RDBMS session ID* – the session ID in the database,
- *Authentication level* – the level of user authentication (Mediator, RDBMS or OS),
- *Login name* – a username entered during logging in,
- *RDBMS username* – the name of database account that a user is logged into (for RDBMS level of authentication that name is the same like *Login name*),

- *Network protocol* – the transmission protocol of a network communication between an application and Mediator server,
- *Workstation address* – a user's workstation address (IP or IPX),
- *RDD login timestamp* – the exact time and date of logging in for a particular user,
- *Total number of opened tables* – the number of tables (working areas) opened by an application or a session, including:
  - – *In shared mode* – tables in a shared mode,
  - – *In exclusive mode* – tables opened in an exclusive mode,
- *Total number of FLOCKed tables* – the number of tables locked by FLOCK() command.

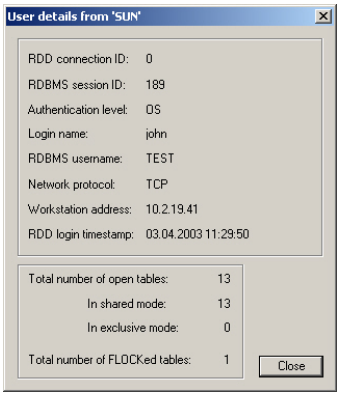


Figure 16. Dialog „*User details from...*"

Button *Close* closes the dialog.

List of locked records

The dialog „*Table ... rlocks on...*" is displayed after pressing the button *Get table RLOCKs* in the window *Tables & locks*. It displays a list of locked records in a selected table.

- *Table* – a table name,
- *Owner* – a table owner (an account in the database),
- *SWA* – the number of a working area of a particular user,
- *Recno* – the number of a locked record,
- *UID* – ID of a Mediator session locking a record,
- *RDD user* – a username (login) of the user associated with the blocking session.

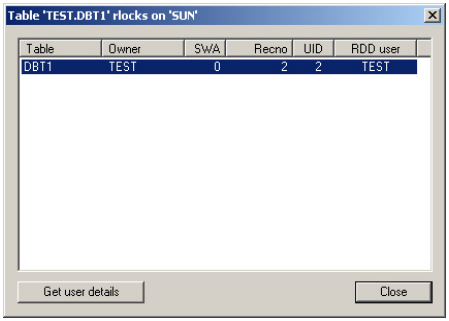


Figure 17. Dialog „*Table ... rlocks on…*"

Buttons:

- *Get user details* – after selecting a row from a list of locks, the window is opened that contains a description of a locking session. (fig. 10). If there are no defined connections on the list, the button *Remove Connection* is inactive.
- *Close* closes the dialog.

Session monitor

A session monitor displays information about all the opened sessions in Mediator server. Each session is related to one application on the client side. The dialog contains a list of sessions, and the data about opened tables for a session marked on the list. Each list row is related to one session and contains:

- *UID* – the session number assigned by the application server during logging in.
- *RDD user* – the name of a user account, where a particular application is logged in.
- *RDBMS user* – the name of a database account, where a particular application is logged in.
- *RDBMS sesID* – a session number assigned by the database server.
- *Proto* – a communication protocol between an application and Mediator server.
- *WS Address* – the address of a workstation where the application is running.
- *BOFL* – seconds elapsed from the last contact of the application with the server.
- *Login timestamp* – the date and time of a user's log-in to Mediator server.

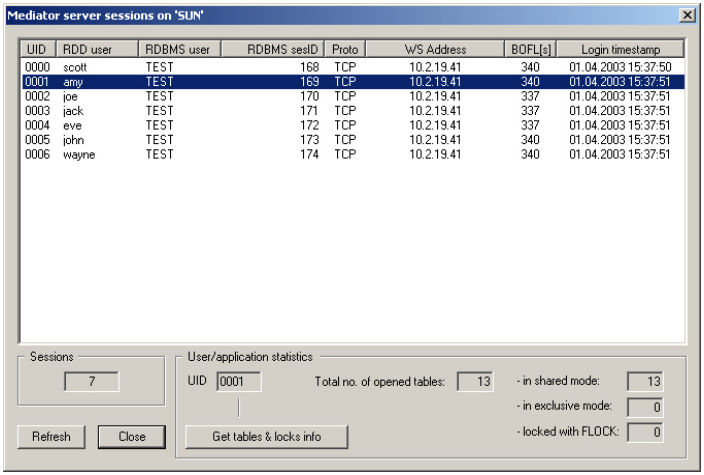


Figure 18. Dialog „*Mediator server sessions on...*"

The remaining data displayed in the dialog:

- *Sessions* – the number of rows on the list (the number of concurrently running sessions).
- *User/application statistics* – this item is related to one, chosen and marked session:
    - *UID* – a session number assigned by the server (a repeated field from the marked row connected to a given session)
    - *Total no. of opened tables* – the number of tables (working areas) opened by a particular user, including:
    - – *in shared mode* – tables in a shared mode,
    - – *in exclusive mode* – tables opened in an exclusive mode,
    - – *locked with FLOCK* – tables locked by FLOCK() command.

- The button *Refresh* refreshes the information displayed in the window.
- The button *Get tables & locks info* opens the dialog *Tables & locks on…* in the context of a selected session. This means that all the information about opened and locked tables refer to a selected session or a user only.
- The button *Close* closes the window.

CHAPTER 4

IV. Adapting applications for work with Relational Database Management System

1. RDD drivers: MEDNTX and MEDCDX

The Mediator package offers two RDD drivers.

The MEDNTX and MEDCDX drivers are designed for the fast adaptation of an application for the cooperation with the database server. Thanks to compatibility of the MEDNTX (MEDCDX) driver with the DBFNTX (DBFCDX) one, the process is simple and fast.

2. Planning of application porting

Division into modules

The process of a large application migration can be time consuming. Since XBASE application can use many data sources simultaneously, it is sensible to divide the large application into modules, which can be ported separately.

Designating tables for porting

Since the structure of an application can be heterogeneous, it is worth considering whether some tables should not be left as *.dbf* files (for example, temporary tables). Creating temporary tables on the database server can cause excessive fragmentation of tablespace which, subsequently, can slow down the server.

3. Stages of adapting an application

In order to port an application operating on .DBF and .NTX files into RDBMS environment, some changes in the source code need to be introduced. Those changes can be divided into several stages:

- Stage I – porting an application to the RDBMS environment without using extensions
- Stage II – using RDBMS transaction mechanisms
- Stage III – using SQL extensions available in the Mediator server
- Stage IV – integration of the ported application with other SQL applications

In order to go through stages I – III, it is enough to use the MEDNTX or MEDCDX driver. In order to go through stage IV, it is convenient to use OLEDB/ADO driver for Mediator , unless external applications or SQL tools only read shared data. In such case the MEDNTX or MEDCDX driver is sufficient.

4. Stage I: basic porting of an application

In stage I, shared data are ported to RDBMS, and the application source code is modified so that the application is able to operate on ported data.

a) Data export

The tools delivered in the Mediator package or home-developed tools can be used during data porting. Data porting takes place in several steps:

- planning the division of data among RDBMS users
- porting the structures of tables
- creating indexes
- loading the contents of particular tables

The size of tables stored in RDBMS is unlimited, so DBF tables can be normalized (wide records can be split into several smaller records) before or after exporting to database server.

☝ **WARNING!**

The Oracle7 server can store tables not wider than 254 columns and so the total number of database fields and expression indexes cannot exceed 254 - 2 additional columns (252). In the Oracle8 server, the total number of columns in the table cannot exceed 1000.

☝ **WARNING!**

Due to performance reasons, it is recommended to conform to a given sequence of steps. Creating expression indexes on tables with loaded data is slow and it considerably slows down the process of data porting.

Planning of data allocation

While planning data allocation it can be assumed that *.DBF located in one directory correspond to tables of one RDBMS user account. Depending on the way *.DBF are stored in directories, three cases can be distinguished:

- All files are stored in one directory:
  This is the simplest case – it requires only one account in the database server. This account will contain the tables corresponding to the *.DBF files from the directory containing data.
- Tables are stored in different directories on one level of a directory tree:
  In this case it is necessary to create as many RDBMS users as there are directories with data. Those accounts will contain tables corresponding to files from particular directories.

- The structure of directories with data consists of many levels:
  This case requires flattening of the structure of directories to one level in the tree. After flattening, follow the procedures described in the previous paragraph.

The names of RDBMS users should correspond to the names of directories with data. This allows the references to tables in applications to be unchanged. The last directory name in the path indicates the name of the RDBMS user. The beginning of a path will be ignored. In case of MS SQL server the database name can by specified before the user name (see MedIgnDBN()).

Example

```
USE d:\dbfs\headq\customer.dbf VIA "MEDNTX"
```

The above example opens the table “customer” on the “headq” account (RDBMS user). If the database name recognition was activated by calling MedIgnDbN(.T.), in case of MS SQL server the above example will open the table “customer” belonging to the user “headq”, located in “dbfs” database.

WARNING!

To make applications access the data stored in different RDBMS accounts, appropriate privileges need to be assigned to database users.

Exporting the structure of tables

Structures of tables can be exported with **DBF2MED** utility delivered with the Mediator package. Description of DBF2MED operation can be found in Appendix F on page F-1. DBF2MED allows to port the structure of a given single database or all databases from a directory. Another method of porting structures is writing your own program using the following commands:

```
COPY STRUCTURE
```

or

```
CREATE ... FROM
```

An appropriate conversion should be applied when code pages of the server and CA-Clipper client are different. In DBF2MED program an appropriate type of conversion is set by the command line argument (see Appendix G). In your own loading program the following commands should be used:

```
SET SERVER CODE PAGE
SET CLIENT CODE PAGE
```

Those commands are described in Chapter VII on page VII-8.

When using dbf2med for CA-VO, remember that DBF tables stored in OEM format are migrated to RDBMS as OEM ones, while ANSI tables are migrated as ANSI. This default behavior could be changed by the use of /FO and /FA run-time options.

☝ **WARNING!**

If you choose an option of loading all databases from a given directory when porting data with DBF2MED program, remember that all files but *.NTX and *.DBT (files) in the given directory are treated in the same way as *.DBF files. In order to avoid errors during loading, delete all files that do not contain application data from the directory.

☝ **WARNING!**

When porting data structures remember that names of ported tables or columns in tables must not be the same as reserved database words. Reserved Oracle words are listed in Appendix A on page A-1.

Creating indexes

Indexes, especially expression ones, should be created on ported structures before loading data. It can be done with the utility **NTX2MED** or **CDX2MED** delivered together with the Mediator package. A description of these utilities can be found in Appendix G on page F-4.

Another method of creating indexes is writing your own program using the following command:

```
INDEX ON ...
```

In case of different code pages of the server and client, you should use the conversion as described in „*Exporting the structure* of *tables*" on page IV-3.

☝ **WARNING!**

If expression indexes are created, then regardless of the applied indexing method, the functions and user variables used in indexes need to be linked.

☝ **WARNING!**

When porting indexes remember that their names must not be the same as reserved Oracle words. Reserved words are listed in Appendix A on page A-1.

Loading data

To load data, use the tool **DAT2MED** supplied in the Mediator package. A Description of the DAT2MED program operation can be found in Appendix F on page F-5. The program allows to load data to existing structures. If a directory name is to be a DAT2MED parameter, the program will attempt to read data from all files in that directory with the exception of files with DBT and NTX extensions.

Another method of loading data is using your own loading program that reads record by record from the table opened with **VIA ”DBFNTX” (“DBFCDX”),** and writes the read records to the table opened with **VIA “MEDNTX” (“MEDCDX”)**.

In case of different code pages of the client and server, use the conversion as described in the „*Exporting the structure* of *tables*” paragraph on page IV-3.

☝ **WARNING!**

If expression indexes are created in the table, then regardless of the method of data loading, functions and user variables used in expression indexes need to be linked.

b) Modification of the application source code

Before the XBASE application can connect to the database server and operate on exported data, it is necessary to introduce certain changes in the source code. Those changes are as follows:

- including header file ***mediator.ch***,
- specification of data source (RDD)
- setting the character data conversion mode between the client and server
- changing file functions to their corresponding functions working on RDBMS objects
- avoiding re-indexing and temporary indexes
- eventuall optimization of the method of opening tables
- other modifications

Details of modification steps will be described below. Modifications serving the purpose of increasing security and efficiency of applications will be described in further chapters.

Including header file *mediator.ch*

For CA-Clipper and (x)Harbour

In order to access some extensions offered by the Mediator package, include the mediator.ch file to those *.PRG files which are supposed to use extensions:

```
#include "mediator.ch"
```

It is also necessary to enter the following line in one of source code files:

```
request medntx
```

informing the linker about the necessity of linking the **medntx** module.

IMPORTANT!

The **`request medntx`** command should be specified if the application will use „MEDNTX" driver or „MEDCDX" driver. Functions implementing „MEDCDX" driver are placed in the same library and will be available.

Specification of data source

Reading data from the database server is realized via MEDNTX or MEDCDX driver. The driver can be defined as default with the following command:

```
RDDSETDEFAULT( "MEDNTX" )
```

or:

```
RDDSETDEFAULT( "MEDCDX" )
```

From now on, this command will cause all tables to be created or opened with the MEDNTX (MEDCDX) driver. You can also specify a data source (RDD driver) in case of any command requiring that, for example:

```
USE tab VIA "MEDNTX" SHARED
CREATE tab FROM tab2 VIA "MEDNTX"
```

You can use several drivers simultaneously:

```
USE tab VIA "MEDNTX" SHARED
USE tab2 VIA "DBFNTX" NEW
```

File functions

In most of XBASE applications, there are functions working directly on files, for example:

```
FILE(), DELETE FILE, ERASE, ADIR(), RENAME
```

Since data is not stored in files anymore but in tablespaces of the database server, calling those functions does not make sense. Libraries supplied in the Mediator package contain a range of useful functions replacing file functions of XBASE. The functions corresponding to file functions are listed below. A detailed description of those functions can be found in Chapter VII.

```
FILE()   → MedIsTable()
         → MedIsIdx()
         -> MedIsBag()
         -> MedGetFile()
```

```
DELETE FILE
ERASE,
Ferase()→ DROP INDEX
         → MedDropIdx()
         → DROP TABLE
         → MedDropTab()

ADIR()   → MedGetTabs()
         → MedGetIdxs()
         -> MedGetFile()

RENAME,
FRename()→ MedRenTab()
```

To simplify the adaptation of XBASE software to work with the relational databases as much as possible together with the Mediator client software the MedFileLib library is distributed. The library defines the following functions: *MedFile(), MedAdir() and MedFErase().* These functions are also provided in the form of a source code, so that they can be modified according to the specific needs. In case of CA-Clipper and (x)Harbour the mentioned library is included in mfilelib.prg and medntx.lib files.

By use of the additional functions and commands porting the application becomes much easier. It is enough to locate all the calls referencing files, decide whether in the new version of the application they will apply to the RDBMS objects and, if yes, substitute the commands as follows:

```
FILE()          → MedFile()
DELETE FILE     -> MEDDELETE FILE
ERASE           → MEDERASE
FErase()        -> MedFErase()
ADIR()          → MedAdir()
RENAME          → MEDRENAME
FRename()       -> MedRenTab()
```

Since it is not always possible to mimic the exact behavior of the file operation functions on the RDBMS objects, the resulting behavior of Med... functions might differ from the expected one. We recommend the reader to consult the manual pages of those functions for further information.

☝ **WARNING!**

In the locations where file functions do not operate on databases or indexes, or operate on databases or indexes that will be still opened by the DBFNTX/DBFCDX driver, there is no need of program modification. In those locations, functions will work unchanged.

Avoiding reindexing

In typical XBASE applications, reindexing of databases is a frequent and time consuming operation. In case of applications using Mediator and RDBMS, this operation is unnecessary. All indexes created during loading data (immediately after creating a table structure) are automatically updated, even if they are not active from the point of view of the XBASE application (not activated using SET INDEX TO command). Even in case of a damage of a workstation, there is no danger of loosing the data or index consistency.

When designing or adapting an application to work with Mediator, avoid creating expression indexes on non-empty tables as well. This operation is inefficient in database and, therefore, it is time consuming. All expression indexes should be created on an empty table after creating a table structure. Those indexes will be updated during any data modifications.

Opening tables – optional step

In order to increase the efficiency of an application, tables stored in RDBMS should preferably be opened only once with the USE or **dbUseArea** command, and they should be kept open. Avoid the situation in which a table is opened for one or several operations and then it is closed. An opened table does not compromise the data safety even after turning the workstation or server off unexpectedly. If your application is designed in such a way it opens tables only for the time they are required and then immediately closes them, you should consider this step as optional and implement it only when performance is not satisfactory.

c) Benefits acquired from first stage of adapting an application

The goal of the first stage of adapting an application is starting the XBASE application on data stored in RDBMS at the minimum expense. The application acquires a new quality already after the first stage. The basic benefits are:

- safety of data stored in database management system - consistency of indexes is guaranteed
- capability of operating on huge data sets
- increased efficiency and scalability of the application
- client-server architecture
- capability of operating in the wide area network
- ability to use data functions available in RDBMS
- capability of using a wide range of SQL applications for data analysis (including OLAP).

Further steps of adapting of an application are presented below. Those steps will allow an application to acquire further benefits, including the guarantee of data consistency thanks to transactions, and increase in the efficiency of generating reports with SQL queries.

5. Stage II: the interface to the RDBMS transaction system

a) Introducing transactions

One of the fundamental advantages of advanced database servers, including Oracle, MS SQL Server or SQL Anywhere is the existence of transaction mechanisms. Those mechanisms guarantee saving a requested set of operations as a whole, or in case of an error, rolling back all of them. The Mediator provides the XBASE application an interface to transaction system of the server in the form of three commands:

```
BEGIN TRANSACTION
COMMIT TRANSACTION
ROLLBACK TRANSACTION
```

Those commands designate locations in the program that are critical for data consistency. The instruction BEGIN TRANSACTION starts a transaction. The code fragment which implements the transaction is finished in the moment of using the instruction COMMIT TRANSACTION, which stores changes in the database, or using the instruction ROLLBACK TRANSACTION, which rolls back any changes made from the moment of issuing the instruction BEGIN TRANSACTION. Should a critical error occur in a code fragment between the instruction BEGIN TRANSACTION and the instruction COMMIT TRANSACTION or the execution is aborted due to an external event (such as turning the server off or a hardware defect), the transaction will be rolled back as a whole and the workstation will receive an appropriate message (as long as the connection between the workstation and the server is not broken).

Remarks on transactions

In order to retain the maximum compatibility with XBASE, RDBMS tables served by the MEDNTX or MEDCDX driver contain an additional RECNO column which stores numbers of records. The Mediator server sets RECNO numbers during appending records to the table. If an application inserts a record into the table during a transaction, then other applications inserting records into the same table will be suspended until committing or rolling that transaction back. The consequence is that in case of a wrong sequence of inserting records it is possible to cause a deadlock.

Example

Application 1 inserts a record into table A, while application 2 inserts a record into table B at the same time. Both applications use transaction. If, before committing the transaction, application 1 attempts to insert a record into table B and application 2 attempts to insert a record into table A, then they will cause a deadlock waiting for each other.

In order to avoid the above problem, you can change the default behavior of APPEND BLANK with the command

```
SET APPEND TIMEOUT
```

This command specifies the time of waiting for the record insertion with the command APPEND BLANK. This command is described in detail in Chapter VII on page VII-97.

Some commands can't be executed within a transaction. This especially concerns DDL (Data Definition Language) commands which automatically commit the transaction (for example DROP INDEX ...).

IMPORTANT

The above problem can be avoided by giving up the continuous record numbering. In such a case, many applications can insert records into the same table at the same time within transaction, without the risk of deadlock. You can learn more about the non-continuous record numbering mode from the description of the following commands/functions:

```
SET PERFORATED NUMBERING ON/OFF
MedSetPerf()
MedPerfMod()
MedPerfRC()
```

b) Benefits acquired from stage II of adapting an application

After introducing transactions, the main benefit is guaranteed consistency of data. Sudden defects or turning the workstation or server off, even during the execution of a transaction, will not compromise safety of databases. Application fragments, which are responsible for deleting flawed operations from tables, can be removed. Other transaction systems formerly used are not necessary anymore.

6. Stage III: using SQL extensions

The Mediator software allows the user to access data stored in RDBMS via SQL language. Using SQL queries requires a programmer to possess basic knowledge of this language, but the final results, which include the increase in efficiency of an application, data safety and reducing network traffic, are worth additional effort.

a) Introduction of SQL

The SQL language can be used in several different ways:

USE AS ”select ....”

The instruction USE AS ”select ....” allows to open the SQL query in a similar way as the table, and its results can be viewed in the same way as contents of the table. They can be saved on the disk in the temporary or permanent table as well. A detailed description of the USE ... command can be found in Chapter VII on page VII-102.

MedExecSQL(”...”)

The **MedExecSQL** function allows to execute practically all SQL statements with the exception of ‘SELECT...’ queries. A detailed description of that command can be found in Chapter VII on page VII-22.

MedSelVal(”...”)

The **MedSelVal()** function can be used for convenient execution of SQL statements which return only one value.

MedSqlPar(p1)

The **MedSqlPar()** function can be used to specify parameter values for SQL statements used by the above function/commands.

SQL procedures stored on the server

Using procedures that are stored on the server results in a significant increase in efficiency. Procedures written in SQL language with procedural extensions (in Oracle it is PL/SQL, in MS SQL Server – Transact-SQL) are stored on the database server and executed with the command **MedExecSQL** described in the previous paragraph. Using Oracle PL/SQL procedures is described in Appendix D on page D-1.

Filters

SQL conditions can be used during data filtering. Traditional filters set in XBASE applications are relatively ineffective. The Mediator for Oracle offers the possibility of setting CA-Clipper filters on the server. Conditions of filters are translated into

SQL and they are applied on the server. There is the possibility of setting special filters in which conditions are specified directly in SQL language. Such conditions are not translated, but sent directly to the server.

The description of setting filters and command line syntax can be found in paragraph *„Fil”* on page V-3.

b) Benefits acquired from stage III of adapting an application

Stage III, i.e. entering SQL into application should be treated as its further development. It is difficult to assess that process as finished at any given point in time. Generally it is common to find fragments of programs that are more or less operation critical and which, therefore, should be replaced with SQL commands. With the increasing fraction of code being based on SQL, benefits are more and more visible. They are primarily:

- reports previously created with SKIP are produced much faster when created with SQL;
- complicated calculations can be located on the server which reduces the load of the workstation;
- network traffic is significantly reduced.

7. Stage IV: integrating an application with other SQL applications

Applications created with SQL tools (for example Designer/Developer 2000, Delphi, Power Builder and others) can read data from the Mediator tables created with the MEDNTX (MEDCDX) driver. Such tables can be modified if certain security measures are taken. When creating the SQL application, the following guidelines need to be followed:

- updating contents of *recno* and *is_deleted* fields
- updating expression indexes correctly (ie$0, ie$1 ...)
- record locks placed by the XBASE application using RLOCK() function **outside a transaction** will not be visible in SQL applications
- table locks placed by XBASE applications using FLOCK() function will not be visible in SQL applications

If you need to modify Mediator-managed tables from GUI applications running in Windows environment (other then Harbour or xHarbour ones) the most convenient and safe method is to deploy OLEDB driver for Mediator. This driver allows you to access Mediator tables via ADO database API. ADO/OLEDB driver for Mediator automatically maintains all required additional columns in Mediator tables and ensures correct cooperation of Mediator and ADO locks.

8. Portability

The XBASE/Mediator application can work with different database systems, i.e. the same executable can be started with Oracle, MS SQL Server7, PostgreSQL. MySQL or SQL Anywhere. In order to achieve that goal ensure following requirements to be met:

- Use ANSI SQL in SQL statements (all given manufacturers declare compatibility with the standard).
- While writing SQL conditions that test empty fields use *MedNulChar* and *MedNulDate* functions that return values representing null characters and dates depending on the database.
- Convert dates to the representation readable for particular database system using *MedSrvDate* function in SQL queries.
- Since every manufacturer delivers his own function set, using them is a problem because of lack of standards. Therefore, depending on which database server is used, use different functions in SQL queries. The *MedDbCode*, *MedDbName* and *MedDbVer* functions that return number, name and version of database server can be used for detecting database system and optionally for choosing different SQL expressions where necessary.

9. Guidelines for Clip-4-Win users

Clip-4-Win application should be linked with

MNTXCW52.LIB, MEDQBW.LIB and optionally NOAUTLOG.OBJ (for Clipper 5.2)

or

MNTXCW53.LIB, MEDQBW.LIB and optionally NOAUTLOG.OBJ (for Clipper 5.3).

Sample Clip-4-Win/Mediator application with link scripts can be found in Mediator client installation in SOURCE\SAMPLE\CLIP4WIN directory.

If the application is linked without NOAUTLOG.OBJ file, then all Mediator connection parameters (MEDNTEADDR, MEDNODEADDR, MEDSOCKET, MEDUSER, MEDPASSWD, MEDCS) should be set before running the application. If the application is linked with NOAUTLOG.OBJ file then MedLogin() function should be used. There is no possibility to enter user name and password on the automatic login screen like in a text application.

Other guidelines on adapting CA-Clipper applications to work with Mediator could be used also for Clip-4-Win applications.

V. Extensions of the MEDIATOR package

1. Using objects owned by other users

The Mediator by default uses tables stored on the account of the RDBMS user which username is given during setting a connection. The Mediator version 1.3 and later is able to use objects stored on other users accounts. This capability pertains to all commands and functions in which the name of the table is specified, the only exception being **MedRenTab** function, which has to be executed by the owner of the table. The necessary condition of successful execution of such an operation is that current RDBMS user has to have appropriate privileges to use objects owned by another user. The list of privileges required to perform particular operations can be found in "APPENDIX C" on page C-1. The name of the table stored on another user's account ought to be preceded by an appropriate username and \ (*backslash*), and handed over to the function. For example:

```
USE TEST\TABLE1 VIA "MEDNTX"
```

The command opens the table TABLE1 stored on the user's account TEST.

Full file specification with a drive letter can be used. The file name will be used as the table name, and the last directory name will be used as the username. The rest of the file specification will be ignored. For example:

```
USE C:\KAT1\TEST2\TAB.DBF VIA "MEDNTX"
```

The server will attempt to open the TAB table stored on the account TEST2.

The capability of using objects owned by other users is especially useful if the application uses *DBF databases with the same names located in different directories. The directory structure can correspond to the set of RDBMS users.

In case of MS SQL Server, it is possible to specify the name of the database before the user name. In the above example the specified database is "kat1". Before database name could be recognized, you have to call MedIgnDBN(.F.) function.

2. Using non-standard extensions in names of databases and indexes

The Mediator version 1.4 and higher allows to use non-standard extensions of databases and indexes. The default extension for a database is .DBF, while for an index it is .NTX. Extensions of those types are omitted, i.e. the table stored on the

database server, corresponding to the TEST.DBF database, is to be named TEST. The non-standard extension can be used for a database or index. For example, if table TEST.ABC is to be created, then the corresponding RDBMS table is named TEST_ABC (the dot is replaced with the underline character). In the CA-Clipper program the TEST.ABC name is still used, for example:

```
USE TEST.ABC
```

The corresponding RDBMS table name to be used in SQL expressions is TEST_ABC, for example:

```
USE x AS "SELECT column1 FROM TEST_ABC" NEW
```

3. Cooperation of MEDNTX and MEDCDX drivers

Regardless which of the two drivers was used to create the table, the Mediator makes it possible to open that table with either MEDNTX or MEDCDX driver. However, remember that opening the table with MEDNTX driver, the application will be able to access only to indexes created using that driver. While using the MEDCDX driver, the application can only use indexes created with MEDCDX driver. Regardless which of the two drivers was used for creating indexes, **all indexes are always updated**, even if they were not activated from the level of XBASE application (with SET INDEX TO command).

4. The scope mechanism (SCOPE)

The Mediator libraries contain set of functions enabling using scope mechanism. Those functions allow specifying the lowest and/or the highest value of index key present within the scope of visible database records. During working on DBF databases, using scope mechanism was possible only in certain RDD drivers compliant with DBFCDX. The Mediator can specify scopes both on indexes created with MEDNTX driver as well as on those created with MEDCDX driver.

For purpose of handling scopes the Mediator uses following functions:

MedSetScpe() - setting or deleting scope
MedClrScpe() - deleting scope

Parameters and operation of MedSetScpe() function is compliant with CA-Clipper ORDSCOPE() function. In applications using scopes with ORDSCOPE() function calls it is enough to replace those calls with MedSetScpe() calls.

The MedClrScpe() function was introduced for purpose of better compliance with Sx_SetScope() and Sx_ClrScope() functions used in SIX driver produced by

SuccessWare Inc. company. If the application adapted for the Mediator uses those functions, replace as follows:

Sx_SetScope() -> *MedSetScpe()*
Sx_ClrScope() -> *MedClrScpe()*

In case of the application using following commands:

```
SET SCOPETOP TO
SET SCOPEBOTTOM TO
SET SCOPE TO
CLEAR SCOPE
```

the replacement can be executed automatically by adding file *medscope.ch* (#include ”medscope.ch”) on the beginning of *.PRG file.

5. Deleting objects from the database

Tables and indexes stored in the RDBMS are not like .DBF or .NTX files. It is not possible to use functions that delete files, such as ERASE or DELETE FILE in order to remove them. In order to delete a table or index, you should use special functions of the MEDIATOR package:

```
DROP TABLE <xcTableName>
DROP INDEX <xcIndexName>
```

A detailed description of commands and corresponding functions can be found in Chapter VII.

6. Filtering

Two commands are implemented to make the use of filters more effective. Filtering of records is possible to be executed either on a client workstation or on the server. As regards time, it is more effective to filter records on the server since fewer records need to be transmitted over the network. In order to toggle between filtering modes, the following command can be used:

```
SET FILTERING ON SERVER
```

or

```
SET FILTERING ON CLIENT
```

Filtering is set on the server as default. If filtering is chosen on the server then Clipper condition (filter) is translated into corresponding SQL expression (only for

Oracle). Sometimes it is not possible to translate condition and filtering records is located on the workstation despite default server filtering mode. The filtering mode can be checked by executing following function after SET FILTER ...

```
MedFltRes()
```

This function returns one of two values defined in the *mediator.ch* file:

OTC_FILTER_ON_SERVER - the filter was created on the server
OTC_FILTER_ON_CLIENT - the filter was created on the client

Accurate translation of Clipper functions into SQL expressions is complicated, and translated expressions are often much longer than the original ones. That's why sometimes it is better to use the SQL filtering expression directly, which facilitates the immediate setting of the SQL condition for the WHERE clause. Such a filter can be defined with the following command:

```
SET SQL FILTER TO <cSQLFilterCondition>
```

For example:

```
SET SQL FILTER TO "check_code between 50 and 100"
```

The SQL filter is always set on the server. Using the filter requires familiarity with the SQL syntax. The syntax is described in appropriate manuals, for example in "ORACLE Server SQL Language Reference Manual" (for Oracle) or "Adaptive Server Anywhere Reference Manual" (for SQL Anywhere).

Sometimes it is convenient to use the name of the column which implements Clipper's expression index in the filter's condition. It is especially useful in places where the expression index depends on the column and you want to use the index while calculating a condition, since a substantial increase of speed can be achieved that way. In such a case use the name of the expression index preceded by the IE$ prefix. For example:

```
SET SQL FILTER TO "IE$IW1 LIKE 'ABC%'"
```

This command allows to obtain only those records whose beginning of the expression index from IW1 index equals "ABC".

The **MedIsFltr**() function checks whether the filter is set.

A detailed description of commands as well as examples can be found in Chapter VII.

WARNING!

If **dbSetFilter** function is used for setting the filter on the server, you should add the second parameter (the condition text) as well. If that parameter is omitted, then the filter will be set on the client. The problem does not exist if the SET FILTER

command is used (it is replaced with the call of **dbSetFilter** function with both parameters).

7. Using SQL

The MEDIATOR package contains extensions enabling the execution of queries (with the USE command) as well as other SQL statements on the database server (**MedExecSQL** function). An extended USE command allows to execute SQL queries directly on the server and use query results as if they were a table (a database). For example:

```
USE qry AS ”SELECT number,name,wage FROM employees” NEW
SCROLLABLE
BROWSE()
```

Results of this sample query can be used just as any other table.

SQL queries can greatly speed up time-consuming calculations (for example, summarizing or grouping), and they also can help to avoid using other areas (*join query*), as well as simplify many other complicated functions.

The correctness of opening the *query* can be checked with **NETERR()** function. If the function returns .T., then it is possible to check by using **MedCmdRes** function, whether or not the reason of failure is a database error.

The **MedExecSQL** function allows to execute any SQL command (with the exception of a query), and especially to create or call functions stored on the server (*stored procedures*). A complete example demonstrating how to use the procedure package stored on the Oracle server can be found in Appendix D on page D-1.

In order to use queries in a correct way, you ought to know SQL. A detailed description of SQL can be found in documentation (for Oracle - *„ORACLE SQL Language Reference Manual”).*

Testing empty fields in the CA-Clipper table

In RDBMS tables corresponding to a Clipper databases, unspecified values (NULL) should not be stored. That is why IS NULL or IS NOT NULL comparison should not be used in SQL queries. Depending on the field type, undefined values are represented by:

- for character and logical fields – one space
- for numerical fields – 0
- for dates – the first day in Julian calendar (Oracle) or January 1, 1 AD (other servers)

Appropriate SQL conditions should be formed as follows:

```
"WHERE character_field = "+MedNulChar()
"WHERE date_field = "+MedNulDate()
```

Alternatively:

```
WHERE character_field = ' '
WHERE date_field = to_date(1, 'J') /* Oracle */
```

The first example is more portable.

The appropriate condition for numerical fields:

WHERE numerical_field = 0

Detailed command descriptions and examples can be found in Chapter VII.

8. Trapping SQL errors

The execution errors of SQL can be serviced in two modes. The following command causes displaying of each SQL error on the screen and aborting the program:

```
SET SQL ERROR VERBOSE
```

The command:

```
SET SQL ERROR SILENT
```

allows to check the correctness of SQL command execution by **MedCmdRes** function. When **MedCmdRes** function returns the database error code, its text can be retrieved with **MedErrText** function.

A detailed description of functions as well as examples can be found in Chapter VII.

9. Transactions

In order to ensure the logical consistency of data, the operations that modify data should be executed within a transaction. Using transactions ensures that all operations belonging to the transaction will be executed or rolled back.

The start of transaction is designated with the command:

```
BEGIN TRANSACTION
```

The transaction can be finished with the commit command:

```
COMMIT TRANSACTION
```

or the roll back command:

```
ROLLBACK TRANSACTION
```

If a program begins a transaction but it does not finish it with COMMIT TRANSACTION or ROLLBACK TRANSACTION, the transaction is rolled back even if the program was terminated correctly. In case of any error encountered during the execution of transaction, the transaction will be rolled back.

The success of BEGIN TRANSACTION, COMMIT TRANSACTION, ROLLBACK TRANSACTION commands can be checked by **MedTrRes** function which returns the TRANS_SUCCESS or TRANS_ERROR result. Checking the result should be made immediately after executing the command.

During execution of other commands within a transaction (for example APPEND, UPDATE), errors can be encountered and transaction can be rolled back. Where appropriate, call **MedTrRes** function, which in this case will return one of the following values:

TRANS_SUCCESS	- the operation was successful;
TRANS_ROLLBACK	- the operation was unsuccessful, the transaction will be rolled back;
TRANS_TIMEOUT	- the operation was unsuccessful due to the lack of access to some resources and it will be rolled back

If one workstation inserts records to a particular table, then other workstations which want to insert records to that table as well wait until that transaction finishes (that is, until it performs the commit or roll back operation). It is due to the necessity of setting the unique number (RECNO) of inserted records.

To prevent the danger of dead-locks resulting from waiting for the transaction completion one may give-up the sequential RECNO numbering (see SET PERFORATED NUMBERING ON/OFF) or introduce append timeout using SET APPEND TIMEOUT command.

A detailed description of commands and examples can be found in Chapter VII.

WARNING!

If an application enforces rolling the transaction back using the ROLLBACK TRANSACTION command, then it is responsible for releasing all locks in databases and records placed by RLOCK(), DBRLOCK() and FLOCK() functions. If the transaction is rolled back due to disconnection of the client and the server, then all locks are removed automatically and the client is logged off from the MEDIATOR server.

10. The record marking subsystem in RDBMS

For the effective use of SQL queries from Clipper, it is necessary to mark those records in RDBMS on which a query will operate. The MEDIATOR contains a set of functions for creating tables of markers. The marker table contains either RECNO numbers of marked records, or RECNO and values of a given field from marked records. Indexes which allow to find a record quickly are created automatically. The marker tables can be used for checking whether or not a record of a given RECNO number (of a given field value) is marked (EXISTS clause), or in queries joining many tables.

Handling errors while using marking functions is the same as during SQL queries. In order to mark records, the following functions can be used:

MedMrkNew
MedMrkOpen
MedMrkAdd
MedMrkDel
MedMrkAll
MedUMrkAll
MedMrkFlush
MedMrkClose
MedMrkRemv
MedMrkNum

A detailed description of functions as well as examples can be found in Chapter VII.

11. Specification of storage parameters for tables and indexes in Oracle

The Oracle database gives its users the capability of controlling the space occupied by tables and indexes. During the creation of the table or indexes, a number of other parameters that define tablespace can be specified, such as the initial extent, next extent, percentage free and percentage used. These parameters are as follows:

- TABLESPACE – a tablespace name
- PCTFREE – the maximum block filling ratio in percentage points
- PCTUSED – the minimum block filling ratio in percentage points (for tables only)
- INITIAL – the size of first extent allocated for an object at the moment of its creation
- NEXT – the size of the second extent allocated for the object (if the first one is filled)
- PCTINCREASE – the increase of subsequent extents allocated for an object expressed in percentage points

- MINEXTENTS – the minimum number of extents allocated to the object
- MAXEXTENTS – the maximum number of extents

The Mediator client allows to specify all of the above parameters with the following functions:

```
OraStTbsp
OraStPctFr
OraStPctUd
OraStPctIc
OraStEIntl
OraStENext
OraStEMin
OraStEMax
OraStDeflt
```

Specified parameters are valid for all created tables and indexes until the change or return to default values (**OraStDeflt** function).

More information about Oracle storage parameters can be found in Oracle documentation: *„Oracle7 Server Concepts”*, *„Oracle7 Server Application Developer’s Guide”*, *„Oracle7 Server Administrator’s Guide”*.

A detailed description of functions and examples can be found in Chapter VII.

12. Locking tables and records

After a failed attempt to lock a table or record, CA-Clipper applications often trigger series of renewed attempts (FLOCK, RLOCK, DBRLOCK functions) until achieving a desired effect or preset time out. For the purpose of reducing network traffic during multiple attempts of locking a table or record, the Mediator software was equipped with the capability of locating such an operation on the server. The following commands serve that purpose:

```
SET LOCK TRY
SET LOCK INTERVAL
```

The **SET LOCK TRY** command allows to specify a number of attempts to lock a record or table, while the **SET LOCK INTERVAL** command specifies the time in milliseconds between attempts. Defined parameters pertain to all FLOCK, RLOCK and DBRLOCK commands following them until the moment of defining other parameters. The default is one additional attempt executed after 100 milliseconds. If LOCK TRY and LOCK INTERVAL parameters are not defined, the process of locking is executed as follows:

1. If the client can lock a resource, the lock is set and the client receives a positive response immediately
2. If the resource is locked by another client, then the process of serving the client waits a 100 milliseconds and it checks the possibility of locking a requested resource again
3. If the resource is unlocked, the client locks it and receives a positive response, otherwise it receives a negative response

If the LOCK TRY parameter is greater than 1, then an attempt of locking is repeated many times until getting a lock or reaching a limit of attempts. During the execution of attempts to lock, the client workstation does not send any commands to the server and it does not receive any responses either.

In order to receive the list of locked records use **MedGetLLst** function, or its extended version **MedGetLAll**.

A detailed description of functions and examples can be found in Chapter VII.

13. Mediator client in multithreaded applications

32-bit Mediator client can be used in multithreaded applications. The access to Mediator client functions and MEDNTX/MEDCDX driver methods is strictly sequenced. This means that the simultaneous call of these functions, for example dbSkip() from one thread and MedExecSQL from the second thread ia acceptable, but it will be serialized. Depending on the call order one of the functions will be executed, while the second will be suspended. After completing of the first function the execution of the second function will be resumed.

Certain operations require a call of several different functions. For example, MedSQLPar() function sets the parameters used by the subsequent call to MedExecSQL() function. Since version 5.0.4 of Mediator client such sequences are also safe in a multithreaded environment, because each thread has its own memory for this type of context parameters. In particular, it applies to the following functions:

dbOpen
dbCreate
dbOrderCreate
MedSelVal
MedExecSQL
MedLobNew
USE .. AS ‘SELECT ..’, USE .. AS PROC, USE ... AS FUN

The above functions can have parameters set by the following functions:
MedSetQry
MedSetCurs
MedIdxSQL, MedIdxLen
MedSQLPar, MedSQLParEx, MedSQLParA
MedTabTemp
MedStTbsp
OraStTbsp, OraStPctFr, OraStPctUd, OraStPctIc
OraStEIntl, OraStENext, OraStEMin, OraStEMax, OraStDeflt, OraStDflt
MedPrepStmt, MedExecStmt
MedLogErr, MedLgMsg

Using Mediator client in multithreaded applications is transparent to the application programmer, but it must be insured that the related calls such as MedSQLPar() and MedSelVal() are made in the same program thread. Otherwise, the parameters set by MedSQLPar() are not seen in MedSelVal().

CHAPTER 6

VI. Working with Unicode

About Unicode

Unicode was designed as a standard for representation of text in all (or almost all) languages of the world. Unicode's development is coordinated by Unicode Consortium organization, the most actual information could be found on their Web site (www.unicode.org). Unicode is implemented by different character encodings. The most commonly used are:

- UTF8 – which uses 1 to 4 bytes for a single character,
- UTF16 – always representing a character on two bytes and
- UCS2 – 2 byte encoding, which is an MS Windows standard.

Unicode in databases supported by Mediator

Oracle

Oracle database offers 2 types of encoding: for character columns (base type) (CHAR/VARCHAR2/CLOB) and for national character columns (NCHAR/NVARCHAR2/NCLOB). Both types are determined by the administrator during the database creation. First type could be single byte or Unicode, the second one could be only Unicode (AL16UTF16 or UTF8 to choose). Different combinations of character types are possible for example: the base character encoding could be single byte (like WE8ISO8859P1, known as ISO Latin 1) and national character set .Unicode (like UTF8). Or base character set UTF8 (i.e. AL32UTF8), and national character set UTF16 (i.e. AL16UTF16).

Columns of type LONG are able to store characters encoded in a base database character encoding.

MS SQL Server

MS SQL Server is able to store Unicode characters only in columns of type NCHAR/NVARCHAR/NTEXT encoded in UCS2 standard.

MySQL

MySQL can store Unicode characters encoded in UTF8 and UCS2 standards. The default character set encoding is determined during the database creation (for example "create database db_name character set utf8"). It is possible to use other then default encoding for a column by specifying different encoding during a column creation ("create table my_table (col_ucs2 varchar(50) character set ucs2").

PostgreSQL

The method of encoding characters for PostgreSQL database is specified during the database creation. Unicode characters encoded in UTF8 standard could be stored in the database created with option encoding ‘UTF8’ (create database db_utf8 encoding ‘UTF8’).

Using Unicode in applications working with Mediator server

16-bit applications (Clipper) working with Mediator server on any database do not have Unicode support.

32-bit applications (Harbour/xHarbour) in the future will be able to present and store Unicode data using standard RDD drivers (DBFNTX/DBFCDX). For now (half of 2011) Unicode support is not complete, only functions converting Unicode to/from single byte character sets are available. Our goal is giving users the possibility to use Unicode now and in the future, when Unicode support will be complete. Currently Harbour/xHarbour applications can take advantage of an extended Unicode support in Mediator. It is executed in the following way (by example of data storing):

- text data encoded in a default single byte application code page (one set by HB_SETCODEPAGE()) are passed to MEDNTX/MEDCDX driver

- MEDNTX/MEDCDX driver converts data from an application code page (for example CP852) to the server code page (for example UTF8).

- data in server encoding are passed to Mediator server and there stored to the database with no conversion applied (and no information loss)

Reading data proceeds in a similar way:

- request of data reading is send from the client application to Mediator server

- data in server encoding are read by Mediator

- read data is sent to the client application without any conversion

- MEDNTX/MEDCDX driver converts the read data to the client default code page and passes data to the application.

This type of work requires that Unicode is fully supported by a database client software, which is used by Mediator. This feature is fully implemented by Oracle and MS SQL Server. ODBC software available for PostgreSQL has Unicode version, so this version of driver should be used (additional notes are placed below). ODBC driver for MySQL does not support Unicode.

By properly changing internal code page xHarbour/Harbour applications can store to the database characters from different languages. The application can work using different languages preserving integrity of character data stored in the database in

Unicode encoding. It is also straightforward to have applications using different languages (codepages) working on the same UNICODE database.

The other possibility is to work on character data encoded in the database in Unicode standard from an application working with a single-byte character set and take advantage of conversions build-in RDBMS software. Then default conversions will be applied, they usually give the desired result. Storing data is proceeded in the following way:

- character data encoded in the default single byte application code page are passed to MEDNTX/MEDCDX driver
- MEDNTX/MEDCDX driver performs character conversion according to the client and server code page set by MedSetCPC/ MedSetCPS or HB_SETCODEPAGE() functions
- single-byte character data is sent to Mediator server and there (as single byte) is written do the database via database interface used by Mediator server (which is OCI for Oracle, ODBC for other databases). At this moment conversions build in the database client software are applied – these conversions as a result allow to store data in Unicode encoding.

Using this mode all applications (16- and 23-bit) can work on all database servers supported by Mediator.

Unfortunately, this approach limits you to only one single-byte codepage at a time. This is because all character data sent from applications to Mediator must arrive in the same single-byte codepage configured for database client. Consequently, it is not possible to easily implement multilingual applications this way

xHarbour/Harbour applications can use the following functions to benefit from the extended support for Unicode:

- MedSetDefColEnc() – function allows to specify the default encoding of character data for newly created tables or newly added columns
- MedSetSqlEnc() – function allows to set the default text encoding for SQL statements
- MedLogin(,,,,,[, <cCharset>]) – additional optional cCharset parameter allows to set the Unicode support mode on login. For Oracle and for PostgreSQL server it means the possibility to work on data stored in the base Unicode character set.
- MedSelStrVal() – function to select character values in arbitrary encoding
- MedSQLParEx()/MedSqlParA() – functions allow specifying SQL statements character parameters in arbitrary encoding

Below the extended Unicode support for various database servers is described.

Oracle

Extended Unicode support is available for both the base (CHAR/VARCHAR) and the national (NCHAR/NVARCHAR) character set. The support for the base character set should be set when connecting to the database (MedLogin). Work on the national character set (nvarchar2/nclob) can be enabled by calling the MedSetDefColEnc() function with the appropriate parameter (e.g. STRENC_UTF16).

Warning!

Since the LONG type columns can store only data in the base character set, it is not possible to enable national character set for MEMO fields of LONG type (MedMemType(MED_MEMO_LONG)). So, for example, for a database with a base character set WE8ISO8859P1 and national character set AL16UTF16 the following settings are wrong:

```
MedSetDefColEnc( STRENC_UTF16 )
MedMemType(MED_MEMO_LONG)
dbCreate("test_tab",{{ "FM", "M", 0, 0 }})
```

while the following is possible:

```
MedSetDefColEnc( STRENC_UTF16 )
MedMemType(MED_MEMO_CLOB)
dbCreate("test_tab",{{ "FM", "M", 0, 0 }})
```

MS SQL Server

The default mode is to work on single-byte character sets. Calling MedSetDefColEnc function with STRENC_UCS2 parameter before creating a table or adding a column will cause the character fields will be created as nvarchar/ntext and will be transferred to/from the client as UCS2.

MySQL

Work on the Unicode data is possible by using the ODBC driver built-in conversion. Mediator server reads Unicode data using ODBC driver, which converts it to single-byte data. There is no possibility to send Unicode data (UFT8, UCS2) to MySQL server, because the ODBC driver (MySQL ODBC 3.51 or 5.1 by MySQL AB) does not allow this.

When operating on data stored in Unicode UTF8 we recommend the use of MySQL server version 1.5.1914 or higher (previous versions have problems with the conversion).

The best way is to create a database with Unicode character set and create tables in this database:

create database database_name character set utf8 collate here_the_collation_you_choose

or:

create database database_name character set ucs2 collate here_the_collation_you_choose

The list of possible settings for collation parameter is available in the MySQL documentation. It could be for example: utf8_general_ci (which is the default value for utf8) or ucs2_bin (for ucs2).

PostgreSQL

Working on Unicode character data is possible in two modes:

1. using the ODBC driver that supports Unicode (Unicode PostgreSQL); transferring UTF8 fields from a client application to Mediator server is possible
2. using the ODBC driver that does not support Unicode, then the data is sent as a single-byte characters and converted to UTF8 by default for PostgreSQL mechanisms.

Work via ODBC driver which supports Unicode allows of course sending single-byte data, in the same way as via ODBC driver that does not offer Unicode support.

WARNING!

When character data is transferred as UTF8 (mode 1) , the data is sent to the server and converted as follows:

- from Mediator client application to Mediator server – encoded in UTF8

- Mediator server performs a conversion form UTF8 to UCS2 and passes the data to ODBC driver

- ODBC driver converts UCS2 data to UTF8 and sends it to PostgreSQL server

Transfer and data conversion in the opposite direction (from PostgreSQL server to the client application) is performed as follows:

- character data from the PostgreSQL server is sent in UTF8 to the ODBC driver

- ODBC driver converts data to UCS2 and sends it to Mediator server

- Mediator server converts data from UCS2 to UTF8 and sends it to the client application.

Double data conversion is done, because ODBC driver supports only UCS2.

Creating a database with UTF8 encoding:

cerate database test_utf8 encoding ‘UTF8’

The table below shows the possible settings for database servers and the possible settings for Mediator functions.

Settings other then listed below are not possible and will result in an application error.

Database, possible type of connection (from (x)Harbour client)	**Default database char type and char data transfer mode**	**Possible MedSetDefColEnc() settings and corresponding char data transfer modes**	**The default encoding and transfer of user SQL statements**	**Possible settings for MedSetSqlEnc function**
Oracle **Default characterset AL32UTF8** **National characterset: AL16UTF16** **Connection CHARSET_UNICODE**	Fields VARCHAR2, Transferred as UTF8	• STRENC_DFLT i STRENC_UTF8 Fields VARCHAR2, transferred as UTF8 • STRENC_UTF16 Fields NVARCHAR2, transferred as UTF16	STRENC_UTF8 (transfer UTF8)	• STRENC_UTF8 transfer UTF8 • STRENC_SB transfer SB
Oracle **Default characterset AL32UTF8** **National characterset: AL16UTF16** **Connection CHARSET_SB**	Fields VARCHAR2, Transferred as SB (conversion applied by Oracle to NLS_CHARACTERSET which is set for Mediator server)	• STRENC_DFLT i STRENC_SB Fields VARCHAR2, transferred as SB • STRENC_UTF16 Fields NVARCHAR2, transferred as UTF16	STRENC_SB (transfer SB)	• STRENC_SB transfer SB
Oracle **Default characterset SB, for example. WE8ISO8859P1** **National characterset: AL16UTF16** **Connection CHARSET_SB**	Fields VARCHAR2, transferred as SB	• STRENC_DFLT i STRENC_SB Fields VARCHAR2, transferred as SB • STRENC_UTF16 Fields NVARCHAR2, transferred as UTF16	STRENC_SB (transfer SB)	• STRENC_SB transfer SB

Oracle **Default characterset SB, np. WE8ISO8859P1** **National characterset: UTF8** **Connection CHARSET_SB**	Fields VARCHAR2, Transferred as SB	• STRENC_DFLT i STRENC_SB Fields VARCHAR2, transferred as SB • STRENC_UTF8 Fields NVARCHAR2, transferred as UTF8	STRENC_SB (transfer SB)	• STRENC_SB transfer SB
MS SQL Server **Default characterset SB,** **National characterset: UCS2** **Connection CHARSET_SB**	Fields VARCHAR, Transferred as SB	• STRENC_DFLT i STRENC_SB Fields VARCHAR, transferred as SB • STRENC_UCS2 Fields NVARCHAR, transferred as UCS2	STRENC_SB (transfer SB)	• STRENC_SB transfer SB
PostgreSQL **Default characterset UTF8** **Connection CHARSET_SB**	Fields VARCHAR, Transferred as SB	• STRENC_DFLT i STRENC_SB Fields VARCHAR, transferred as SB	STRENC_SB (transfer SB)	• STRENC_SB transfer SB
PostgreSQL **Default characterset UTF8** **Connection CHARSET_UNICODE** **Necessary using the ODBC driver with Unicode support (PostgreSQL Unicode)**	Fields VARCHAR, Transferred as UTF8	• STRENC_DFLT i STRENC_UTF8 Fields VARCHAR, transferred as UTF8 • STRENC_SB Fields VARCHAR, transferred as SB	STRENC_SB (transfer SB)	• STRENC_SB transfer SB

VII. Functions and procedures of MEDNTX and MEDCDX drivers

This chapter contains the description of additional commands and functions of the MEDIATOR package listed in alphabetical order.

1. BEGIN TRANSACTION

Syntax:

```
BEGIN TRANSACTION [TIMEOUT <nTimeout>].
```

The command designates the beginning of transaction. The optional TIMEOUT clause allows to specify the time in milliseconds of waiting for access to resources locked by other transactions. No time-out specified means unlimited time of waiting for access.

Transaction should be completed by committing it (COMMIT TRANSACTION) or rolling it back (ROLLBACK TRANSACTION). If the program begins a transaction but does not commit it, the transaction will be rolled back regardless of correct termination of program. In case of any error, transaction will be rolled back by the server and SRV/1400 error will be reported. Trapping SRV/1400 error allows to deal with transactions rolled back by server and commence appropriate actions. The transaction is also automatically rolled back in case of losing network connection with the server. Should this happen, NET/1301 error will be reported.

Calling MedTrRes() function and checking the value returned allows to control the correctness of execution of particular transaction phases.

The following operations are **not allowed** during the transaction:

- Creating a table
- Creating an index
- Deleting a table (DROP TABLE)
- Deleting an index (DROP INDEX)
- Purging deleted records (PACK command)
- Deleting table contents (ZAP command)
- Changing a table name (**MedRenTab** command)
- Creating a new marker table (**MedMrkNew** command)
- Marking all records (**MedMrkAll** function)
- Deleting all markers (**MedUMrkAll** function)

- Deleting a marker table (**MedMrkRemv** function)
- Executing SQL commands other than INSERT, UPDATE and DELETE (with the **MedExecSQL** function)

2. COMMIT TRANSACTION

Syntax:

```
COMMIT TRANSACTION
```

The command commits changes made during transaction. Success of COMMIT TRANSACTION command can be checked by calling **MedTrRes** function.

3. DROP INDEX

Syntax:

```
DROP INDEX <xcIndexName>
```

The command deletes *xcIndexName* index associated with the table opened in the active area. The table has to be opened in exclusive mode (USE ... EXCLUSIVE).

Example:

```
USE tab
DROP INDEX ti1
```

After execution of DROP INDEX command, index **ti1** will be deleted.

4. DROP TABLE

Syntax:

```
DROP TABLE <xcTableName>
```

The command deletes the table named *xcTableName*, for example:

```
DROP TABLE tab
```

After the execution of the command, the table named tab as well as all objects associated with it, including indexes, will be deleted. It is not necessary to delete indexes after or before deleting the table.

The function corresponding to the command:

```
MedDropTab( <cTableName> )
```

The function deletes the table named *cTableName*. The string *cTableName* can contain a username preceded by the backslash.

5. MedAdir

Syntax:

```
MedAdir( [<cFilespec>], [<cTabArray>] ) -> nFiles
```

Parameters <*cFilespec*>, <*cTabArray*> are optonal.

<*cFilespec*> specifies the file in a standard way, and can contain both * and ? wild characters. By default, when the <*cFilespec*> option is missing it is assumed to be "*.*". If the path is not given the function executes in the current user account. Otherwise, the last part of the directory name specification is treated as the database user name.

If the function is called without the <*cTabArray*> parameter then it will return the number of the objects corresponding to the given file specification (that is, for the *.dbf files – the number of tables, for *.ntx files – the number of indexes created by the MEDNTX driver and for *cdx files – the number of bags).

<*cTabArray*> specifies the table which should be previously allocated. If the function is called with this parameter then the specified array will be filled with the names of tables, indexes or bags available.

IMPORTANT!

If the extension of the file is not given, the function will return the total number of all tables, indexes (created by the MEDNTX driver) and bags which match the given name.

The function does not return additional attributes of the specified files [<*Sizes*>,<*Dates*>,<*Times*>,<*Attrributes*>] which are available with ADIR() function.

Example

```
numCdx = MedAdir("*.cdx")
IF numCdx>0
    Cdx_names = ARRAY( numCdx )
    MedAdir("*.cdx", Cdx_names)
ENDIF
```

See also: **MedGetTabs(), MedGetIdxs(), MedGetFiles()**

The function is available as the source code in MFILELIB.PRG file and in the medntx.lib library

6. MedChgPwd

Syntax:

MedChgPwd(*<cOldPassword>*, *<cNewPassword>*) -> *nResult*

This function can be used to change the Mediator password of current user directly from application. Please note that only Mediator users passwords can be changed this way (authentication level = Mediator). For RDBMS and operating system authentication methods passwords cannot be changed.

When calling *MedChgPwd()* Mediator user password will be changed to *cNewPassword* only if correct *cOldPassword* is supplied. Function returns following results:

0 – password correctly changed

1 – invalid old password specified - password not changed

2 – not Mediator authentication used – password not changed

3 – unable to save server configuration file – password changed but will revert to old after next Mediator server restart

7. MedChrIdxT

Syntax:

MedChrIdxT(<nCharIndexType>)

The function specifies the character type to be used on the database server for the purpose of character indexes representation. By default, VARCHAR2 type is used.

In very rare situations it may be necessary to use the CHAR field type (the fixed length field) in order to ensure that the key sequence is exactly the same as in Clipper.

MedChrIdxT(MED_IDX_VARCHAR)

Character expression indexes will be created in VARCHAR2 fields.

MedChrIdxT(MED_IDX_CHAR)

Character expression indexes will be created in CHAR fields.

The type designated with **MedChrIdxT** function is used for new indexes until the next function call.

See also: **MedIdxLen**

8. MedClntId

Syntax:

`MedClntId() -> nClientConnectionId`

The function returns the client number assigned by the Mediator server during connection.

9. MedClpComp

Syntax:

`MedClpComp(<lClipperCompatible>) -> lPrevSetting`

Available only in Harbour/xHarbour applications. Function can be useful in cases where Clipper and Harbour/xHarbour applications must work using the same tables via Mediator server.

In version 5.0 of Mediator , many limits resulting from the use of 16-bit Clipper applications have been removed. The transmission buffer has been extended, records wider than 64KB can be used, SQL query and parameters buffer has been extended as well. Some of this changes may result in incompatibilities between Clipper and xHarbour/Harbour applications. For instance if the Harbour application creates table with records wider than 64 KB, Clipper application will be unable to open such a table. *MedClpComp()* function can be used to enforce in xHarbour/Harbour application the same database limits as in the Clipper application. Function

MedClpComp(.T.)

should be called right after connecting to Mediator server, before opening any tables. After such call the RDD driver and Mediator server will prevent xHarbour/Harbour application from performing any database operation that is incompatible with CA-Clipper applications.

See also: **MedLogin**

10. MedClrTbCa

Syntax:

`MedClrTbCa([<cTableName>]) -> lSuccess`

If the table was opened via Mediator server, its description (structure) may stay in Mediator internal memory (cache) until Mediator server is restarted. In such a case,

all subsequent opens get table structure description directly from Mediator memory, not from the database. Then, if the table structure is modified by an external (SQL) application, these changes will not be seen by Mediator applications. *MedClrTbCa()* function can be used to remove table description from Mediator cache, resulting in reading the table structure from database on next table open.

cTableName parameter specifies the name of the table to be removed from Mediator cache. If the table is not open by anybody, its description is removed from Mediator cache and function returns .T. (success). If the table is being used (opened), it cannot be removed from cache and function returns .F.

Calling function without parameters requests Mediator to remove all table entries from its cache. Function will succeed and return .T. if no tables are opened via Mediator at the moment of its execution. Otherwise, function removes from Mediator cache entries off all tables which are not used (opened) but returns .F. as the cache cannot be cleared completely.

11. MedClrScpe

Syntax:

```
MedClrScpe([<nScope>])
```

The function deletes limit defining scope. Optional *nScope* parameter specifies limit to delete. Valid arguments:

0 – the low limit (top scope) - default

1 – the high limit (bottom scope)

Calling this function without arguments is equivalent to calling MedClrScpe(0).

See also: **MedSetScpe()**

12. MedCltVBFx

Syntax:

```
MedCltBFx() -> nClientBugFix
```

The function returns the integer number that represents the small patch number of the Mediator client.

13. MedCltVMaj

Syntax:

```
MedCltVMaj() -> nClientMajorVersion
```

The function returns the integer value that represents the main version number of the Mediator client.

14. MedCltVMin

Syntax:

```
MedCltVMin() -> nClientMinorVersion
```

The function returns the integer number that represents the detailed version number of the Mediator client.

15. MedCltVPth

Syntax:

```
MedCltVPth() -> nClientPatchVersionASCII
```

The function returns the ASCII code of the letter representing Mediator client version modification. This can be a code of ‘ ‘ (space) for base version or a code of letter ‘i’ to ‘p’ representing the modification level.

16. MedCltVSub

Syntax:

```
MedCltVSub() -> nClientSubVersion
```

The function returns the integer number that represents the patch number of the Mediator client.

17. MedCmdRes

Syntax:

MedCmdRes() -> *nSqlResult*

After the call of **MedExecSql** function or after the use of USE ... AS ”select ....” command, **MedCmdRes** function returns OTC_CMD_SUCCESS if execution of the

last command was successful. Otherwise, the RDBMS error code is returned (according to the appropriate documentation, for Oracle - *„Oracle Server Messages and Codes Manual”*). In order to use this function for the purpose of checking the result of execution of SQL command, the following command should be executed:

```
SET SQL ERROR SILENT
```

18. MedColAdd

Syntax:

```
MedColAdd( <cTableName>, <aFieldParams> ) -> lResult
```

The function adds columns to the *cTableName* table. The *aFieldParams* table contains parameters of columns that should be added. The description of each column contains four values:

aFieldParams[N][1] – the name of the added column

aFieldParams[N][2] – the type of the column (C,L,M,N,D)

aFieldParams[N][3] – the length of the column

aFieldParams[N][4] – the precision of the column (0 for non-numerical)

aFieldParams[N][5] – optional – type of character data representation for this column (0 for non-character data). Character data encoding type is used only in xHarbour/Harbor applications – it can be one of the following values:

- STRENC_DFLT – default value – database type used for character columns will be selected based on the *MedSetDefColEnc()* function settings. If not available, the type of connection from application to Mediator server will be used (see *MedLogin*()). In case of CHARSET_SB connection the column will store single byte characters, in case of CHARSET_UNICODE connection the default database type representing UNICODE characters will be used.
- STRENC_UTF8 – character columns will store UNICODE characters encoded as UTF8
- STRENC_UTF16 - character columns will store UNICODE characters encoded as UTF16
- STRENC_UCS2 – character columns will store UNICODE characters encoded as UCS2

If the type of character data is not specified, the STRENC_DFLT is assumed.

If columns has been successfully added, the function returns .T., otherwise it returns .F. The table name can be preceded by the username and backslash character. The table to which columns are added with *MedColAdd* function must not be opened!

Example:

```
fldArr = array(2,4)

* add chr_fld as CHARACTER(50)
fldArr[1][1] = "chr_fld"
fldArr[1][2] = "C"
fldArr[1][3] = 50
fldArr[1][4] = 0

* add num_fld as NUMERIC(10,3)
fldArr[2][1] = "num_fld"
fldArr[2][2] = "N"
fldArr[2][3] = 10
fldArr[2][4] = 3

* add columns
? MedColAdd( "table1", fldArr )
```

See also: **MedSetDefColEnc()**

19. MedColDel

Syntax:

```
MedColDel( <cTableName>, <cColumName> ) -> lResult
```

Function deletes from table *cTableName* column *cColumName.* If successful, function returns .T., otherwise it returns .F. The table should not be open by any user. If the column being deleted is included in the index created from Clipper or (x)Harbour level it will not be deleted and function returns .F. Delete such indexes before using this function.

20. MedColRes

Syntax:

```
MedColRes( <cTableName>, <aFieldParams> ) -> lResult
```

The function changes parameters of the *cTableName* table columns. The *aFieldParams* table contains new parameters of columns that are to be modified. The description of each column contains three values:

aFieldParams[N][1] – the name of the column, whose parameters are to be changed
aFieldParams[N][2] – new length of the column
aFieldParams[N][3] – new precision of the column (or 0 for non-numerical)

If the change of parameters is successful, the function returns .T., otherwise it returns .F. The name of the table can be preceded by the username and backslash character. The table whose columns are to be modified with *MedColRes* function must not be opened!

Changes of column parameters can be a subject to additional limits associated with the database server. For example, the Oracle server does not allow reducing the column length if the table contains records.

WARNING! If parameters of a column used in expression indexes are changed, then these indexes need to be re-created.

Example:

```
fldArr = array(2,3)

* change chr_fld to CHARACTER(50)
fldArr[1][1] = ”chr_fld”
fldArr[1][2] = 50
fldArr[1][3] = 0

* change num_fld to NUMERIC(12,2)
fldArr[2][1] = ”num_fld”
fldArr[2][2] = 12
fldArr[2][3] = 2

* resize columns
? MedColRes( ”table1”, fldArr )
```

21. MedConCS

Syntax:

```
MedConCS() -> cConnectString
```

The function returns the value of parameter MEDCS used for connection to Mediator server.

22. MedConNetA

Syntax:

```
MedConNetA() -> cNetworkAddress
```

The function is available for 32-bit clients only. Returns the value of parameter MEDNETADDR used for connection to Mediator server.

23. MedConNode

Syntax:

```
MedConNode() -> cNodeAddress
```

The function is available for 32-bit clients only. Returns the value of parameter MEDNODEADDR used for connection to Mediator server.

24. MedConSock

Syntax:

```
MedConSock() -> cConnectionSocket
```

The function is available for 32-bit clients only. Returns the value of parameter MEDSOCKET used for connection to Mediator server.

25. MedDate

Syntax:

```
MedDate() -> dServerDate
```

The function returns the system date from the machine where Mediator server is running.

See also: **MedTime(), MedDateTm()**

26. MedDateTm

Syntax:

```
MedDateTm( @dServerDate, @nServerTime ) -> NIL
```

The function returns the system date and time from the machine where Mediator server is running. The date is returned as date and the time is returned as number of seconds elapsed since midnight. Use the following expressions to obtain the hours, minutes and seconds of the current time respectively: INT(nServerTime/3600), INT(MOD(nServerTime,3600)/60) and MOD(nServerTime,60). Function parameters should be passed by reference.

Always use MedDateTm() in case you wish to obtain both server date and time. If you use functions MedTime() and MedDate() their results can be inconsistent in case date has changed between the two function calls.

See also: **MedTime(), MedDate()**

27. MedDbCode

Syntax:

```
MedDbCode() -> nMedDatabaseCode
```

The function returns code of database server to which client is connected to. Values returned:

1 - Oracle

2 - Microsoft SQL Server

3 - Sybase Adaptive Server Anywhere

7 – MySQL

8 – PostgreSQL

9- IBM DB2

0 - no connection to the database server

28. MedDbConn

Syntax:

```
MedDbConn() -> nDatabaseConnects
```

The function returns the number of applications (EXEs) being connected to the Mediator server. This value is identical to the number of connections Mediator opens to database server.

See also: **MedMaxLic()**

29. MedDbName

Syntax:

`MedDbName() -> cRDBMSName`

The function returns the name of the database server to which the client is connected to. If there is no connection to the database, null string is returned.

30. MedDbsUser

Syntax:

`MedDbsUser() -> cMedUserName`

The function returns a username for a given client. If the user identification is executed by the Mediator server, the function returns a RDBMS username corresponding to ID of the Mediator user. If the access authorization is executed on the database server, then the user's ID, that has been given during the program execution, is returned (the same that is returned by **MedRddUser** function).

31. MedDbVer

Syntax:

`MedDbVer() -> cRDBMSVersion`

The function returns the version of the database server to which the client is connected to. If there is no connection to the database, the function returns the null string.

32. MedDelNow

Syntax:

`MedDelNow( [<lRefreshPosition>] )`

Function permanently deletes current rekord. Permanently deleted record is removed from table and cannot be restored later. The table automatically becomes "perforated" that is a table with non-continuous record numbering. < *lRefreshPosition* > parameter specifies if automatic position refresh should be performed.

The default parameter value is .F. which causes the deleted record to remain current record after deletion. If .T. is specified as parameter, the position is automatically updated by moving to the next table record.

See also: **MedSetPerf**

33. MedDiscTm

Syntax:

```
MedDiscTm( <nDisconnectTimeout> )
```

The function specifies the time after which the Mediator server will disconnect and log off the workstation that lost communication with the server due to the network damage or turning the power off. The *NDisconnectTimeout* argument specifies time in seconds and it can have values within the range 20-10000.

The function with APP_KEEPALIVE_DEFAULT parameter resets the disconnection time to default:

```
MedDiscTm(APP_KEEPALIVE_DEFAULT)
```

The following command turns client disconnection mechanism off:

```
MedDiscTm(APP_KEEPALIVE_OFF)
```

34. MedDropIdx

Syntax:

```
MedDropIdx( <nWorkarea>, <cIndex> [,<cBagName>] )
```

The function deletes index *cIndex* of the table opened in the area *nWorkarea.* If the table contains indexes created with MEDCDX driver, specify the name of the order bag as third argument. In such case the second argument needs to be the order name. Names of index and bag can contain the path, which will be ignored. The name of the index can contain extension.

In case of the MEDCDX driver the alternative method of deleting orders is using ORDDESTROY() from CA-Clipper library function.

Example:

```
USE test_tab VIA ”MEDNTX”
IF MedIsIdx(  SELECT(), ”TT_IDX” )
   MedDropIdx( SELECT(), ”TT_IDX” )
ENDIF
```

```
USE test_tab VIA ”MEDCDX”
IF MedIsIdx(  SELECT(), ”ORDER1”, ”BAG1” )
   MedDropIdx( SELECT(), ”ORDER1”, ”BAG1” )
ENDIF
```

35. MedDropTab

Syntax:

```
MedDropTab( <cTableName> ) -> lResult
```

If the function has been executed successfully, it will delete *cTableName* table and return the boolean value .T. or .F. in the opposite case. The *cTableName* can contain username preceded by the backslash character. For example:

```
? MedDropTab( ”test\tab” )
```

After the function is executed, the table named *tab* owned by a *test* user, as well as all objects associated with it, including indexes, will be removed. There is no need to delete indexes before or after deleting the table. In the example, the boolean value, that specifies the success of executing the function, will be displayed.
The command corresponding to the function:

```
DROP TABLE( <xcTableName> )
```

36. MedErrText

Syntax:

```
MedErrText() -> cRDBMSErrorText
```

If **MedCmdRes** returns a RDBMS error code, the **MedErrText** returns the corresponding error text.

37. MedExecSQL

Syntax:

```
MedExecSQL( <cSQLStatement> ) -> nCount
```

The instruction executes SQL command given as the argument. Given all appropriate privileges, it is possible to execute all SQL commands with **MedExecSQL** function, with the exception of SELECT. If INSERT, UPDATE or DELETE command is the

function argument, the function will return the number of processed records. Otherwise, the function will return 0.

For explanation how parameters can be specified for SQL statements, please see description of *MedSqlPar()* function.

WARNING!

All instructions that manipulate tables that correspond to Clipper databases should be executed with special care. UPDATE, INSERT and DELETE commands will be executed even if another user opened the table in the exclusive mode or lock it (i.e. the table) with FLOCK(). The modification of table records with UPDATE command can be executed even though another client workstation locked those records with RLOCK command. Also, after execution of DELETE command of SQL type, the physical sequence of records is not guaranteed (RECNO() function will encounter the gaps in numeration).

In order to modify the tables that correspond to a Clipper database, the OTC strongly recommends using the Clipper language functions (APPEND, DELETE, CREATE INDEX and other). However, there are no obstacles for manipulating other tables, particularly in case of creating them, inserting and deleting records in them, and reading them with USE AS SELECT command (see: Using SQL queries).

38. MedExecStm

Syntax:

MedExecStm(<*nStmtHandle*>) -> result

Executes SQL statements previously prepared using *MedPrepStm()* function. *nStmtHandle* is the statement handle returnem by *MedPrepStm()* function. Typically, application use *MedExecStm()* function to execute the prepared statement many times for different parameter sets defined using *MedSqlPar().*

In case executed statement is a SELECT statement, it should fulfill the same conditions as statement passed to *MedSelVal()* function, .i.e. it should return a single value. This value will be returned as *MedExecStm()* function result. In case SELECT statement returns more than one record, the value from the first record will be returned. If SELECT selects no records, NIL value will be returned.

If the statement in not a SELECT statement, the number of processed records will be returned.

For explanation how parameters can be specified for SQL statements, please see description of *MedSqlPar()* function.

See also: **MedPrepStm(),MedFreeStm(), MedSqlPar(), MedSQLParA()**

39. MedExitFun

Syntax:

```
MedExitFun( [<lExecuteExitFun>] ) -> lPreviousSetting
```

When Clipper or xHarbour/Harbour application terminates, virtual machine automatically executes all functions and procedures declared as EXIT (see Clipper documentation for details). Mediator libraries use *MedStop()* exit procedure to release the connection to Mediator server (if still present) and cleanup the internal library structures. This procedure is automatically executed on application termination, so that you never need to call it yourself.

The order in which EXIT procedures and functions are executed cannot be fully controlled by programmer. Consequently, there can be situations when MedStop() function is called to early – before other procedures which make use of Mediator connection. Function *MedExitFun()* allows you to change the default behavior and prevent the execution of *MedStop()* function. Calling *MedExitFun(.F.)* prevents execution of *MedStop()* exit function.

If you block execution of *MedStop()* function you need to manually call *MedStop2()* function at the end of your application. This will perform the disconnection and necessary cleanup at the moment of your choice. See below for a typical sequence required when manually stopping Mediator library.

Example

```
MedExitFun(.F.)      && turn off the default behavior
. . .                && application code
MedStop2()           && perform cleanup
```

40. MedExRecc

Syntax:

```
MedExRecc( [ <lReccountMode> ] )
```

Function modifies behavior of RECCOUNT() and LASTREC() functions for current workarea. By default (or after calling *MedExRecc(.F.))* table record count is returned directly from Mediator server without the need to access database. In this mode value returned by RECCOUNT() or LASTREC() function will not be accurate if an

external (non-Mediator) application is appending records to a table opened via "MEDNTX" ("MEDCDX") driver.

After calling *MedExRecc(.T.)* every successive call of RECCOUNT() or LASTREC() function for a given workarea will determine the current number or records in table by consulting database, though providing the accurate result. Such a method places additional load on database (especially when using browse() and dbedit() functions) and should be used only when really required.

See also: **MedRfsRecc(), MedLMRecc(), MedLMGRecc()**

41. MedFErase

Syntax:

```
MedFErase( <cFile> ) -> lSuccess
```

<cFile> is the full file specification together with the extension. If the full path is not specified, the function runs on the current user account. Otherwise, the function is executed on objects owned by the user specified in the last part of the directory name specification.

The function will delete the object equivalent to the specified file (that is, for a .dbf file a table will be removed, for .ntx – an index created by the MEDNTX file and for .cdx – a bag created by the MEDCDX driver). Return value is zero (0) if the operation was successful or –1, when an error occurred.

IMPORTANT!

If the extension of the file is not specified, the function will delete the matching object (table, index created by the MEDNTX driver, bag) only when there is exactly one object matching that name.

Example

```
? MedFErase( "test\bag_name.cdx" )
```

After the above call the existing indexes created by the MEDCDX driver and stored in a bag file *bag_name.CDX* on the user *test* account will be deleted (the current user must have enough privileges to perform this operation).

In the case when the table object is deleted, all associated objects (including indexes) will be deleted automatically. There is no need to delete indexes before or after deletion of the table.

See also: **MedDropTab(), MedDropIdx()**

The function is available as a source code in MFILELIB.PRG/MedVOLib.AEF file and in medntx.lib library.

42. MedFile

Syntax:

```
MedFile(<cFilespec>)->lExist
```

<*cFilespec*> is the standard file name specification, it can contain wild characters "*" and "?". If the full path is not specified, the function runs on the current user account. Otherwise, the function is executed on the account of a user specified in the last part of the directory name specification.

The function returns value .T. if the object equivalent to the specified file exists (that is, for a .dbf file – a table, .ntx file – an index created by MEDNTX driver, for a .cdx file – a bag). Otherwise, the function returns .F.

IMPORTANT!

In case when the file extension is not specified the function will check for either table, index (created by MEDNTX driver) or bag with a given name.

Example

```
IF MedFile( "*.cdx" )
   ? " File of type CDX exists!"
ENDIF
```

See also: **MedIsTable(), MedIsIdx(), MedIsBag(), MedGetFiles()**

The function is available as a source code in MFILELIB.PRG/MedVOLib.AEF file and in medntx.lib library.

43. MedFlckInf

Syntax:

```
MedFlckInf( [@<nMedSessId>], [@<nMediatorId>],
[@<nLockType>], [@<dLockDate>], [@<cLockTime>] ) ->
lSuccess
```

The function returns diagnostic information related to the last unsuccessful attempt to lock table (FLOCK) in the current workarea. It can be called only right after

unsuccessful attempt to lock the table. Function returns *.T.* if diagnostic info was successfully read or *.F.* otherwise. When successful, function assigns the following information to the passed by reference function parameters:

nMedSessId – number of the Mediator session which placed the existing lock making it impossible to place the new lock. Each application can obtain its session number using *MedClntId()* function.

nMediatorId – number of the Mediator server connected with the session which placed the existing lock making it impossible to place the new lock. Each application can obtain its Mediator number using *MedMedId()* function. (*nMedSessId*, *nMediatorId*) pair uniquely identifies application working with Mediator server within the farm managed by Lock Manager module.

nLockType – type of the existing lock: MED_LOCK_RLOCK (1), MED_LOCK_FLOCK (2) or MED_LOCK_APPEND (4). MED_LOCK_APPEND means the existing lock has been automatically placed by dbAppend() function.

dLockDate – date (day) when the existing lock was placed

cLockTime – time when the existing lock was placed as a string in the format "HH:MM:SS", 24-hour mode.

Information returned by *MedFlckInf()* function can be used with *MedGetInfo()* function to read the extra information set by the application which placed the existing lock. Alternatively, application can use its own tables containing *nMedSessId* and *nMediatorId* of every application to find the application user who placed the conflicting lock.

See also: ***MedRlckInf, MedSetInfo, MedGetInfo***

44. MedFlocked

Syntax:

```
MedFlocked() -> lResult
```

Function returns .T. if table opened in current workarea was already locked (FLOCK()) by current application. Otherwise function returns .F.

See also: **MedShared()**

45. MedFltRes

Syntax:

```
MedFltRes() -> nResult
```

The function returns the numerical value that represents a filtering method used for execution of the last SET FILTER command. The function returns one of two values defined in *mediator.ch* file:

OTC_FILTER_ON_SERVER - the filter was created on the server

OTC_FILTER_ON_CLIENT - the filter was created on the client

46. MedFTName

Syntax:

```
MedFTName() -> cFullTableName
```

Function returns the full path name of the database opened in current workarea.

See also: **MedTName()**

47. MedFreeStm

Syntax:

```
MedFreeStm( <nStmtHandle> ) -> lSuccess
```

Function releases the statement allocated by *MedPrepStm()* and identified by *nStmtHandle* returnem from *MedPrepStm()*. Every statement prepared using *MedPrepStm()* must be released when no longer needed. The number of prepared statement is limited for every Mediator session.

See also: **MedPrepStm(),MedExecStm()**

48. MedGetFile

Syntax:

```
MedGetFile( <cFileName>, <nFileType>,
[<cFileNameArray>], [<cTableNameArray>] ) ->
nFileCount
```

The function allows to check the existence of an object of type specified by *nFileType* and the name specified by *cFileName* parameter. The object type could be specified using constants defined in *mediator.ch* file as:

- MED_GETF_DBF (value 0) - table
- MED_GETF_NTX (value 1) – NTX index
- MED_GETF_CDX (value 2) – CDX index

The object name (*cFileName*) may include path and wildcards: * and ? The function returns number of objects as specified by parameters.

The parameters *cFileNameArray* and *cTableName Array* (XBASE arrays) are optional and should be allocated before a call to MedGetFile(). An array *cFileNameArray* will be filled by objects' names. If names of the indexes are retrieved (CDX or NTX) then the appropriate elements of the array *cFileNameArray* will be filled with names of the tables, on which the indexes are created.

If the object name does not contain path then current user objects are checked. Otherwise the function checks the objects of the user specified as last directory name in the path.

Example 1

```
* getting names of all tables beginning with DB
* from current user account
numTabs = MedGetFile("DB*",MED_GETF_DBF)
IF numTabs>0
    tab_names = ARRAY( numTabs )
    MedGetFile("DB*",MED_GETF_DBF, tab_names )
ENDIF
```

Example 2

```
* getting the names of all NTX indexes beginning
* with IDX and their tables names from user test
* account
numIdxs =  MedGetFile("TEST\IDX*",MED_GETF_NTX)
IF numIdxs>0
 idx_names = ARRAY( numIdxs )
 tab_names = ARRAY( numIdxs )
 MedGetFile("TEST\IDX*",MED_GETF_NTX, idx_names, tab_names )
 FOR i=1 TO numIdxs
   ? 'Index :',idx_names[i], ' table: ', Tab_names[i]
 NEXT
ENDIF
```

WARNING!

The sign ? is interpreted as any **existing** character – differently then Clipper FILE function (any or no character).

49. MedGetIdxs

Syntax:

```
MedGetIdxs( <nWorkarea>, [<cIdxArray>] ) -> nIdxCount
```

The *cIdxArray* argument is optional.

If the function is called with one argument (number of workarea), the number of indexes opened in that area will be returned. The function returns actual number of all indexes created on the specified table, including indexes that are inactive according to CLIPPER language semantics (activation is done with SET INDEX TO command).

The second function argument can be the name of the table which should be allocated previously. After calling the function with two arguments, the table will be filled with names of indexes of the table opened in area *nWorkarea*.

Warning! In case of MEDCDX driver the function reads names of bags created on the table.

Example:

```
USE test_tab
numIdx = MedGetIdxs( SELECT() )
IF numIdx>0
    idx_names = ARRAY( numIdx )
    MedGetIdxs(SELECT(), idx_names)
ENDIF
```

50. MedGetInfo

Syntax:

```
MedGetInfo( nSessId, nMediatorId, [@<cName32>],
[@<cString32>], [@<cString128>], [@<nInt1>],
[@<nInt2>], [@<nInt3>] ) -> lSuccess
```

The function allows application to read values of session parameters set by the same or other application by calling *MedSetInfo()* function. You need to pass the data uniquely identifying the session which parameters you want to read: Mediator session number (*nSessId*) and Mediator server number (*nMediatorId*). Session and Mediator number can be obtained for instance from *MedRlckInf()* or *MedFlckInf()* functions. If

session parameters were successfully read, function returns *.T.* and their values are assigned to the appropriate passed by reference function parameters. In case of failure, function returns *.F.* If the parameter is missed from the parameter list, its value is not assigned.

cName32 – max 32-character string
cString32 – max 32- character string
cString128 – max 128- character string
nInt1 – 32-bit integer value
nInt2 – 32-bit integer value
nInt3 – 32-bit integer value

See also: ***MedSetInfo, MedRlckInf, MedFlckInf***

51. MedGetLAll

Składnia:

```
MedGetLAll( [<nWorkarea>] ) -> aLockArray
```

The function returns a two-dimensional array of record numbers locked in table opened in area *nWorkarea*. If *nWorkarea* parameter is skipped, the array of locks existing in table opened in current workarea is returned. Each array row contains locked record number, Mediator server ID and the ID of the session which keeps the record locked.

Example:

```
PROCEDURE PrintAllLocks()
  LOCAL aList
  LOCAL nSize
  LOCAL nCount
  aList := MedGetLAll()
  nSize := LEN( aList )
  ? "Locked records: "
  FOR nCount := 1 TO nSize
       ? "Record:", aList[ nCount ][ 1 ]
       ? " MedID:", aList[ nCount ][ 2 ]
       ? " SesID:", aList[ nCount ][ 3 ]
       ?
  NEXT
```

```
  ?
RETURN
```

52. MedGetLLst

Syntax:

```
MedGetLLst( [<nWorkarea>] ) -> aLockArray
```

The function returns a table of record numbers locked in area *nWorkarea*. It corresponds to DBRLOCKLIST() function and it should be used instead. If *nWorkarea* parameter is skipped, a table of locks existing in current workarea is returned.

Example:

```
PROCEDURE PrintCLocks()
  LOCAL aList
  LOCAL nSize
  LOCAL nCount
  aList := MedGetLLst()
  nSize := LEN( aList )
  ? "Locked records: "
  FOR nCount := 1 TO nSize
       ?? aList[ nCount ]
       ?? SPACE( 1 )
  NEXT
  ?
RETURN
```

53. MedGetSLst

Syntax:

```
MedGetSLst() -> aSessionArray
```

The function returns a table of active session numbers on the current Mediator server.

Example:

```
PROCEDURE PrintActSes()
  LOCAL aList
  LOCAL nSize
```

```
   LOCAL nCount
   aList := MedGetSLst()
   nSize := LEN( aList )
   ? "Active sessions on current Mediator server:"
   FOR nCount := 1 TO nSize
        ?? aList[ nCount ]
        ?? SPACE( 1 )
   NEXT
   ?
RETURN
```

54. MedGetSNum

Syntax:

```
MedGetSNum(<cDbaLogin>,<cDbaPwd>,<lAsSysDba>,<cSid>,
<cSerialNumber>) -> nSessionNumber
```

Only for Oracle. The function returns the Mediator session number for an Oracle session identified by SID: cSid and a serial number: cSerialNumber. The DBA username (*cDbaLogin* and password (*cDbaPwd* is required to complete this request. If lAsSysDba is set to .T. then login 'as sysdba' is performed. The function returns the session number on success or NIL on failure.

55. MedGetTabs

Syntax:

```
MedGetTabs( [<cTabArray>], [<cRDBMSUserName>] ) ->
nTabCount
```

The *cTabArray* and *cRDBMSUserName* parameters are optional. *cRDBMSUserName* specifies the database username to whom the function pertains. If the name is not given, the function operates using the currently logged user name.

If the function is called without *cTabArray* argument, the number of tables that is available on the current or specified user account will be returned. The *cTabArray* specifies the table that should be allocated earlier. If the function is called with *cTabArray* argument, then the given table will be filled with names of available tables (databases). If the table is smaller than the number of databases (tables) on the database account, it will be filled completely and the function will return the actual number of databases (tables).

Example:

```
numTabs =  MedGetTabs()
IF numTabs>0
    tab_names = ARRAY( numTabs )
    MedGetTabs( tab_names )
ENDIF
```

56. MedHdSqlFl

Syntax:

```
MedHdSqlFl( <lHideSQLFilterText> )
```

Function allows to specify whether filter text specified in SET SQL FILTER should be returned by DBFILTER() function. By default DBFILTER() returns the filter text. After calling MedHdSqlFl with .T. parameter DBFILTER will return the empty string. By calling MedHdSqlFl with .F. parameter program returns to default settings. You should call MedHdSqlFl(.T.) if the program tries to evaluate SQL filter on the client. In such a case you receive an error: BASE 1449 Syntax error: &. The problem appears for example if user function in DBEDIT sets SQL filter.

57. MedIdxDesc

Syntax:

```
MedIdxDesc( [<nOrderNumber>] ) -> lIndexDescending
```

The function reports whether the specified index is the descending index. Optional *nOrderNumer* argument specifies the index number in the current work area. The function called without an argument checks the current index. The function returns.T. if the index is descending, otherwise .F. is returned.

58. MedIdxKey

Syntax:

```
MedIdxKey(<cTableName>,<cIndex>[,<cOrderBag>])->
cIndexkey
```

The function returns the index key for index *cIndex* of table *cTableName*. If the key of index created with MEDCDX driver is read, you should specify the name of order bag file as the third argument, and the second argument is the order name. If index does not exist, null string is returned.

59. MedIdxLen

Syntax:

```
MedIdxLen( <nIndexLength> )
```

The function specifies the width of the character field for the next created index. It is recommended to use the function before creating a character expression index whose length is not possible to be calculated by Mediator. The length set by the function call is used only while creating next index. After that Mediator returns to the default algorithm of length calculation.

Example:

```
USE tab VIA "MEDNTX"
MedIdxLen(30)
INDEX ON field1+USERFUN(field2) TO ind
```

The index will be implemented in Oracle in the VARCHAR2(30) field (in MS SQL Server in VARCHAR(30) field). Usually you will not need to use **MedIdxLen** function.

See also: **MedChrIdxT**

60. MedIdxLmt

Syntax:

```
MedIdxLmt( [<nMaxWorkareaIndexes>] )
```

Function allows to set the limit of the number of indexes in the single workarea.

By default it is possible to create up to 20 indexes for each workarea (active or inactive). This function allows to modify the limit and create up to 100 indexes in each workarea. The new limit applies only to the workareas which are opened after the limit is set, so it is recommended to call MedIdxLmt() before any workareas are opened. Calling MedIdxLmt() without parameters resets the limit to the default of 20 indexes.

61. MedIdxSQL

Syntax:

```
MedIdxSQL( <cIndexUpdateSQL>>, [<lOraIdxVirtual>] )
```

This function enables you to define the SQL expression which will be used to accelerate the creation of the next index/order. If you specify the SQL expression, Mediator will use it to fill the new column created for the new index. This way a single UPDATE statement will be issued which fills index values for all table records. By default, Mediator server fills the index values on the record-by-record basis. It is important to carefully prepare SQL expression so that it exactly corresponds to normal xBase expression which will be used later for all updates and inserts.

Beginning from version 5.0.4.0 MedIdxSQL function has second optional parameter, with default value of .F., valid only for Oracle 11 or newer. If the parameter is set to .T. then the column storing the values for an expression index is created as virtual, with the SQL value of cIndexUpdateSQL parameter. Each time when a column value is accessed it is calculated by Oracle server with using of given SQL expression (please see the Oracle documentation on virtual columns).

Attention! This function should be called right before calling the function which creates index.

Example 1:

```
USE tab VIA ”MEDNTX”
MedIdxSQL(“to_char(id,’999999999’)+to_char(date,’YYYYMMDD’)”)
INDEX ON STR(id,10)+DTOS(date) TO ind
* The above index will be created using specified Oracle expression
* During normal operation the specified xBase expression will be
* used to update index key values
```

Example 2:

```
USE tab1 VIA ”MEDNTX”
MedIdxSQL("RPAD(C1,40)||to_char(N1,'999999999')",.T.)
INDEX ON C1+STR(N110,0) TO ind2
* The column storing the value of the expression index will be
* created as a virtual column using specified Oracle expression.
* Each time accessing the column value it will be calculated by
* Oracle server.
```

62. MedIdxUniq

Syntax:

```
MedIdxUniq( [<nOrderNumber>] ) -> lIndexUnique
```

The function informs whether a specified index is of UNIQUE type. The optional *nOrderNumber* argument specifies the index number in the current work area. The

function called without an argument checks the current index. The function returns .T. if the index is unique, otherwise it returns .F.

63. MedIgnDbN

Syntax:

```
MedIgnDbN( <lIgnoreDatabaseName> )
```

This function works only with Microsoft MS SQL Server 7/2000.

The function sets the way the path specified in USE, MedGetTabs(),... commands is interpreted. If the parameter is .T. (the default setting) then the path is interpreted as default, that is the last directory specified is considered as RDBMS user name. Parameter .F. instructs Mediator to use the directory name specified before the user name as the name of database where the table is located.

Example:

```
USE db1\user1\tab1 via "MEDNTX"
    && opens tab1 of user1 from default database
MedIgnDbN(.T.)  &&
    && opens tab1 of user1 from db1 database
USE db1\user1\tab1 via "MEDNTX"
```

See also: **MedIgnPath()**

64. MedIgnPath

Syntax:

```
MedIgnPath( <lIgnorePath> )
```

The function sets the way the path specified in USE, MedGetTabs(),... commands is interpreted. If the parameter is .F. then the path is interpreted as default, that is the last directory is considered as RDBMS user name. Parameter .T. directs Mediator to ignore specified path and open the table of the currently logged user.

See also: **MedIgnDbN()**

65. MedIsBag

Syntax:

```
MedIsBag( <nWorkarea>, <cBagName> ) -> lResult
```

or

```
MedIsBag( <cTableName>, <cBagName>] ) -> lResult
```

The function can be used only on indexes created with the tables handled by the MEDCDX driver. The function returns **.T.** for the table opened in area *nWorkarea*, or the table *cTableName* if bag *cBagName* exist, otherwise **.F.** is returned. The name of the bag can contain extension and path, which will be ignored.

Example:

```
USE test_tab VIA ”MEDCDX”
IF NOT MedIsBag( SELECT(), ”TEST_TAB” )
    INDEX ON pole1 TAG order1 TO TEST_TAB
    INDEX ON pole2 TAG order2 TO TEST_TAB
    INDEX ON pole3 TAG order3 TO TEST_TAB
ENDIF
```

See also: **MedIsIdx()**

66. MedIsFltr

Syntax:

```
MedIsFltr( <nWorkarea> ) -> lFilterInUse
```

The function returns .T. if a regular or SQL filter is active for the table opened in *nWorkarea*.

67. MedIsIdx

Syntax:

```
MedIsIdx(<nWorkarea>,<cIndex> [,<cBagName>])->lResult
```

or

```
MedIsIdx(<cTableName>,<cIndex> [,<cBagName>])->lResult
```

The function returns **.T.** if there is index *cIndex* for the table named *cTableName* or opened in area *nWorkarea*. Otherwise, **.F.** is returned. If there are table indexes created with MEDCDX driver, specify the name of order bag file as third argument and order name as second argument. The name of index and bag can contain path, which will be ignored. The name of the index can contain extension.

Example:

```
USE test_tab VIA ”MEDNTX”
IF NOT MedIsIdx(  SELECT(), ”TT_IDX” )
    INDEX ON pole1 TO TT_IDX
ENDIF

USE test_tab VIA ”MEDCDX”
IF NOT MedIsIdx(  SELECT(), ”Order1”, ”BAG1” )
    INDEX ON pole1 TAG order1 TO BAG1
ENDIF
```

See also: **MedIsBag()**

68. MedIsTable

Syntax:

```
MedIsTable(<cTable>) -> lResult
```

The function returns **.T.** if the table (a database) named *cTable* exists in a database, otherwise **.F.** is returned. The table name *cTable* can contain a username followed by \ (slash) character.

Example

```
IF MedIsTable( ”TEST_TAB” )
    tab_count++
END IF
```

69. MedIsTr

Syntax:

```
MedIsTr() -> lIsTransaction
```

The function returns **.T.** when program is performing a transaction, otherwise **.F.** is returned.

70. MedKeyCnt

Syntax:

```
MedKeyCnt( [<lRawFlag>] ) -> nRecCount
```

Function returns a number of table records regarding the set scope boundaries and (eventually) server (SQL) filters. There must be an active order set before calling this function.

MedKeyCnt(.T.) – default behavior – SQL filters are **not** considered when counting records

MedKeyCnt(.F.) – SQL filters are considered when counting records

See also: **MedKeyNo(), MedKeyGoTo()**

71. MedKeyCtCa

Syntax:

```
MedKeyCtCa( [<lLogCacheOn>] ) -> lPrevLogCacheState
```

By default, Mediator RDD driver use special optimization algorithm and cache for calculating logical number of records. This saves many unnecessary references to Mediator server and SQL database. If you experience problems with logical number of records you can switch off the optimization using *MedKeyCtCa()* function. Function returns previous status of the optimization algorithm (on/off).

MedKeyCtCa(.T.) – turns on the optimization (default behavior)

MedKeyCtCa(.F.) – turns off the optimization – logical number of records is counted by sending the command to Mediator and SQL database.

Optimization of counting the logical number of records is especially important when browsing large databases with active index.

See also: **MedKeyCnt(), MedKeyNoCa()**

72. MedKeyGoto

Syntax:

```
MedKeyGoto( <nNewPosition> ) -> lSuccess
```

Function performs GOTO commant to the recod identified by a logical position *nNewPosition*. Logical position takes into account current order, scope and server (SQL) filters.

In xHarbour version, the same operation can also be executed using *OrdKeyGoto()* function.

In Harbour you can also use *DbOrderInfo(DBOI_POSITION,nNewPosition).*

Other functions operating on logical record numbers:

OrdKeyNo(), DbOrderInfo(DBOI_KEYNO) – returns logical number of the current record regarding server filters and scope settings.

DbOrderInfo(DBOI_KEYNORAW) – returns logical number of the current record regarding scope settings. Does not take into account server or client filters.

OrdKeyCount(), DbOrderInfo(DBOI_KEYCOUNT) – returns number of records in logical order regarding server filters and scope settings.

DbOrderInfo(DBOI_KEYCOUNTRAW) – returns number of records in logical order regarding scope settings. Does not take into account server or client filters.

See also: **MedKeyNo(), MedKeyCnt()**

73. MedKeyNo

Syntax:

```
MedKeyNo( [<lRawFlag>] ) -> nLogicalRecno
```

Function returns a logical record number of the current record in current order. Record number calculation regards the set scope boundaries and (eventually) server (SQL) filters. There must be an active order set before calling this function.

MedKeyNo(.T.) – default behavior – SQL filters are **not** considered when computing logical record number

MedKeyNo(.F.) – SQL filters are considered when computing logical record number

See also: **MedKeyCnt(), MedKeyGoTo()**

74. MedKeyNoCa

Syntax:

```
MedKeyNoCa( [<lLogCacheOn>] ) -> lPrevLogCacheState
```

By default, Mediator RDD driver use special optimization algorithm and cache for calculating logical record numbers. This saves many unnecessary references to Mediator server and SQL database. If you experience problems with logical record

numbers you can switch off the optimization using *MedKeyNoCa()* function. Function returns previous status of the optimization algorithm (on/off).

MedKeyNoCa(.T.) – turns on the optimization (default behavior)

MedKeyNoCa(.F.) – turns off the optimization – all functions that return logical record number will send the command to Mediator and SQL database.

Optimization of finding logical record numbers is especially important when browsing large databases with active index.

See also: **MedKeyNo(), MedKeyCtCa()**

75. MedLgMsg

Syntax:

```
MedLgMsg(<nLoginErrorCode>) -> cLoginErrorText
```

The function returns error message corresponding to the error code *nLoginErrorCode* returned by MedLogin() function.

See also: **MedLogErr(), MedLogin()**

76. MedKillSes

Syntax:

```
MedKillSes(<nSesId>,<cDbaLogin>,<cDbaPwd>>,
<lAsSysDba>) -> nErrorNumber
```

The function sends a request for Mediator server to kill a session identified by *nSesId*. To complete the request the server needs also database system user login name *cDbaLogin* and the password *cDbaPwd*. If *lAsSysDba* is set to .T. then login 'as sysdba' is performed. The function returns 0 on success or value greater than 0 on failure.

77. MedLibErr

Syntax:

```
MedLibErr() -> nErrorNumber
```

If an error occurred when calling functions: *MedLibLoad() or MedLibFree()* then the function *MedLibErr()* will return a specific code number which corresponds with the *GetLasErr()* WIN32 function. The function should be called immediately after the call to the function which caused the error.

MedLibExec() function does not set the code of the last error.

See also: **MedLibLoad(), MedLibFree(), MedLibExec(), MedLibExe2()**

78. MedLibExec

Syntax:

```
MedLibExec(<nLibHandle>,<nFunID>,<cParams>) -> cResult
```

The function sends to Mediator server a request of executing the function number *nFunID* from the DLL library loaded previously with the a call to *MedLibLoad()* and identified by the handle *nLibHandle.* The remote function will receive the *cParams* buffer as an input parameter. The value returned by the function loaded from the DLL will be returned as *cResult.*

The maximum size of the passed parameters cannot exceed 62000 bytes. The maximum size of the returned value is 7KB or is equal to the value given when loading the DLL library with a call to *MedLibLoad()* (max. 62000 bytes).

Example

```
params = ""
result = MedLibExec(libhdl,1,params)
?? "MedLibExec() returned ", LEN(result)
? " bytes of data"
for i = 1 to len(result)
  ? "DATA["+STR(i,2)+"] = ",ASC(SUBSTR(result,i,1))
next
wait
```

See also: ***MedLibExe2(),*** **MedLibLoad(), MedLibFree(), MedLibErr()**

79. MedLibExe2

Syntax:

```
MedLibExe2(<nLibHandle>,<nFunID>,
          @<cCmdBuffVar>, <nCmdBytes>,
```

```
@<cResultBuffVar>, <nMaxResultBytes>)
                                -> nResultBytes
```

The function sends to Mediator server a request of executing the function number *nFunID* from the DLL library loaded previously with the a call to *MedLibLoad()* and identified by the handle *nLibHandle*. The remote function will receive *nCmdBytes* of data from *cCmdBuffVar* buffer as an input parameter. Function returns the number of bytes returned by a remote (DLL) function. Returned data is stored in *cResultBuffVar* buffer. *nMaxResultBytes* parameter specifies the size of the result buffer.

cCmdBuffVar and *cResultBuffVar* parameters have to be passed by reference (buffer variables should be prefixed with @ operator), in exactly the same way as in FREAD() function from CA-Clipper library.

The maximum size of the passed parameters cannot exceed 62000 bytes. The maximum size of the returned value is 7KB or is equal to the value given when loading the DLL library with a call to *MedLibLoad()* (max. 62000 bytes).

Example

```
parBuff = ""
retBuff = space(100)
resLen =
MedLibExe2(libhdl,1,@parBuff,LEN(parBuff),@retBuff,LEN
(retBuff))
?? "MedLibExe2() returned ", resLen
? " bytes of data"
for i = 1 to resLen
  ? "DATA["+STR(i,2)+"] = ",ASC(SUBSTR(retBuf,i,1))
next
wait
```

See also: ***MedLibExec(), MedLibExe2(),*** **MedLibLoad(), MedLibFree(), MedLibErr()**

80. MedLibFree

Syntax:

```
MedLibFree(<nLibHandle>) -> nResult
```

The function requests the Mediator server to unload the dynamically loaded library (DLL) identified by the *nLibHandle*. The function returns 0 if the operation was successful or 1 if an error occured. When the function returns 1 the specific error code can be obtained by calling the *MedLibErr()* function.

Example

```
rval = MedLibFree(libhdl)
IF rval == 1
   ? "MedLibFree error, LAST ERROR=",MedLibErr()
   QUIT
ENDIF
? "Library unloaded ok"
```

See also: **MedLibLoad(), MedLibExec(), MedLibExe2(), MedLibErr()**

81. MedLibLoad

Syntax:

```
MedLibLoad(<cLibraryName> [,<nMaxRetData>]) ->
nLibHandle
```

The function requests the Mediator server to load the DLL library with a name specified as *cLibraryName* and informs at the same time that the maximum size of the data returned by functions from that library does not exceed *nMaxRetData*. The function returns a unique handle (identifier) in the range of 1-8. If the loading was not successful, then the function returns 0 and the specific error code can be obtained by calling the *MedLibErr()* function. Up to 8 libraries can be used in parallel by a single client application. Note that the *cLibraryName* parameter specifies the name of the DLL with the optional path as it is seen on the Windows NT/2000 with Mediator server. If the path to the library is not given, the library should reside in one of the directories specified by the Windows system path.

The *nMaxRetData* parameter is optional and doesn't need to be given if the size of the returned data will not exceed 7KB. The maximum value of this parameter is 62000 bytes. There is no need to specify the maximum size of the arguments passed to the functions as the software will recognise it automatically when calling the *MedLibExec()* function.

Example

```
libhdl = MedLibLoad("sample.dll",10000)
IF libhdl == 0
   ? "MedLibLoad error, LAST ERROR=",MedLibErr()
   ? "Make sure sample.dll is located in system DLL"
```

```
   ? „path on NT/Mediator server"
   QUIT
ENDIF
? "Library sample.dll loaded OK, handle", libhdl
```

See also: **MedLibFree(), MedLibExec(), MedLibExe2(), MedLibErr()**

82. MedLMCnAct

Syntax:

MedLMCnAct()-> *nLMActiveConnections*

Function returns the number of active (established) network connections between Mediator server and *Lock Manager* module.

See also: **MedLMRecc(), MedLMGRecc(), MedLMMode(), MedLMCSet(), MedLMTGlob()**

83. MedLMCnSet

Syntax:

MedLMCnSet()-> *nLMNumberConnections*

Function returns the number of network connections configured in Mediator server to establish to *Lock Manager* module. This value can be changed by modifying *Options/LM Links* setting in Mediator server options.

See also: **MedLMRecc(), MedLMGRecc(), MedLMMode(), MedLMCnAct(), MedLMTGlob()**

84. MedLMGRecc

Syntax:

MedLMGRecc([<*lReccountMode*>])

Function globally (for all workareas) sets record number synchronization mode in multi-Mediator installations with *Lock Manager* module.

Calling *MedLMGRecc(.T.)* turns global record number synchronization mode ON.

Calling *MedLMGRecc(.F.)* turns global record number synchronization mode OFF (default).

More about global record number synchronization mode you can learn from description of function *MedLMRecc().*

See also: **MedRfsRecc(), MedExRecc(), MedLMRecc()**

85. MedLMMode

Syntax:

```
MedLMMode()-> lLockManagerMode
```

Function returns .T. (true) if Mediator server is cooperating with *Lock Manager* module to make possible globally coordinated table opens. It is true when *Options/Lock Manager/Enabled* option of Mediator server is selected.

See also: **MedLMRecc(), MedLMGRecc(), MedLMCnSet(), MedLMCnAct(), MedLMTGlob()**

86. MedLMRecc

Syntax:

```
MedLMRecc( [ <lReccountMode> ] )
```

Function modifies behavior of RECCOUNT() and LASTREC() functions for current workarea in installations with many Mediator servers and *Lock Manager* synchronization module. By default (or after calling *MedLMRecc(.F.))* table record count is returned directly from Mediator server to which application is connected. In this mode value returned by RECCOUNT() or LASTREC() function may be not accurate because it does not take into account (or takes into account with delay) records appended via other Mediator servers.

Calling *MedLMRecc(.T.)* instructs RECCOUNT() and LASTREC() functions to read record count from *Lock Manager* module which synchronizes all Mediator servers. This value takes into account all record inserted via all cooperating Mediator servers

The most accurate but also the slowest method of determining the number of records in a table is reading it directly from database (see: *MedExRecc()* function).

Global record count synchronization mode places additional load on Mediator and *Lock Manager* module (especially when using browse() and dbedit() functions) and should be used only when really required.

Use *MedLMGRecc()* function to set global record count synchronization mode for all workareas.

See also: **MedRfsRecc(), MedExRecc(), MedLMGRecc()**

87. MedLMTGlob

Syntax:

```
MedLMTGlob()-> lTableGlobal
```

Function returns .T. (true) if table opened in current workarea is opened in global mode coordinated by *Lock Manager* module (*lockman.exe*). Operations performed on the table opened in global mode are visible to all Mediator servers cooperating with *Lock Manager*. Operations performed on tables opened in local mode are visible only to applications connected to the same Mediator server.

Table is opened in global mode if Mediator server which opens it is cooperating with *Lock Manager* module (*Options/Lock Manager/Enabled*) and the appropriate rule is defined for *Lock Manager* (*tabrules.cfg* file in *Lock Manager's* working directory). It is important to remember that table rules are read only on lockman.exe restart.

See also: **MedLMRecc(), MedLMGRecc(), MedLMMode(), MedLMCnAct(), MedLMSet**

88. MedLogAsN2

Syntax:

```
MedLogAsN2(<lLogicalFieldsAsN2>)->lPrevSetting
```

By default, in all tables created via MEDNTX or MEDCDX RDD driver logical fields are represented in SQL server as character fields containing 'Y' – true or 'N' – false. In case other SQL applications are accessing data, it could be preferable to store logical values as numbers in NUMERIC fields.

After calling *MedLogAsN2(.T.),* all subsequently created tables will store logical values as numbers in NUMBER(2) fields. Values stored in such fields will be interpreted as:

0 – false (.F.)

other value – true (.T.)

Calling *MedLogAsN2(.F.)* restores default table creation mode.

Calling *MedLogAsN2(…)* function influences the way MedColAdd() function implements the logical fields.

See also: **MedLogTVal(), MedMemType(), MedColAdd()**

89. MedLogErr

Syntax:

```
MedLogErr(<nLoginErrorCode>)
```

The function raises error corresponding to the error code *nLoginErrorCode* returned by MedLogin() function.

See also: **MedLgMsg(), MedLogin()**

90. MedLogged

Syntax:

```
MedLogged() -> lResult
```

The function returns **.T.** if there is active connection to Mediator and database servers, otherwise **.F.** is returned.

91. MedLogin

Syntax:

```
MedLogin(<cMedNetAddr>, <cMedNodeAddr>, <cMedSocket>,
<cMedUser>, <cMedPasswd>, <cMedCS> [,<cCharset>] ) ->
nResult
```

This function creates a connection with the Mediator and database servers. Function arguments correspond to DOS environment variables for CA-Clipper and (x)Harbour application. Variables are described in chapter "Preparing client". The connection from CA-Clipper application is created automatically if, during the program linking, file *noautlog.obj* is **not** linked. Linking this file means that the program will create the connection itself by executing *MedLogin* function.

Parameters:

cMedNetAddr – network address (corresponds to MEDNETADDR variable)

cMedNodeAddr – node address (corresponds to MEDNODEADDR variable)

cMedSocket - port number (corresponds to MEDSOCKET variable)

cMedUser - RDBMS or Mediator username (corresponds to MEDUSER variable)

cMedPasswd - password (corresponds to MEDPASSWD variable). If global application password is defined in Mediator server, this password should be attached at the end of user password in square brackets such as in userpasswd[apppasswd]

cMedCS - database ID (corresponds to MEDCS variable)

cCharset – optional – (corresponds to MEDCHARSET variable) identifier of the character set the client expects to work with. The possible values are

- CHARSET_SB (default value) which means application will work with a single-byte character set. Text data is exchanged between the database and MEDNTX/MEDCDX driver in a default single-byte database client codepage used by Mediator server. When the application writes or reads data, it is converted from/to the single-byte codepage set in application via HB_SETCODEPAGE() function. The CHARSET_SB setting allows to connect to both database using UNICODE character set and a single-byte character set as well. In case of the UNICODE database, the conversion to the single-byte client codepage is performed by the database client software. CA-Clipper application can use CHARSET_SB setting to connec to and work with UNICODE database.
- CHARSET_UNICODE which means application will work with UNICODE characters provided the database supports UNICODE. Oracle supports UNICODE if database was created with AL32UTF8 character set. For PostgreSQL the database must be created with UTF8 character set and the ODBC driver used must support UNICODE. Text data is exchanged between database and MEDNTX/MEDCDX driver in UNICODE. At the

moment when text data is read/written by the application it is converted to/from the single-byte application codepage set with HB_SETCODEPAGE() function.

If the connection to Mediator server exists at the moment of calling **MedLogin** function then it will be automatically closed before creating a new connection. It is recommended to explicitly close the connection by previously calling **MedLogout** function. In order to check whether the connection exists, use **MedLogged** function. In case of an attempt to open or create a table without the connection, the table will not be opened and NETERR() function will return **.T.** value. Calling any other function that requires the access to the Mediator server (for example, **MedGetTabs**) without the opened connection will display “No connection to Mediator” error message.

If the connection is opened correctly, **MedLogin** function returns 0. The following values designate errors:

1 - "Unable to allocate communication buffers" – no memory is available for communication buffers

2 - "Unable to initialize communication environment" – error during network initialization

3 - "Unable to connect to server. Bad address?" – connection error – most probably a bad address was used

4 - "User name too long"

5 - "Unable to login - Invalid user/password?" - login error – most probably caused by a bad username or password

6 - "MEDIATOR client & server incompatibility" – Mediator server and client versions are incorrect

7 - “Too many server users” – number of licensed simultaneous connections to Mediator server is exceeded

8 - “General Mediator server error” - Mediator server problem (for example, server out of memory)

9 - "Incompatible medqb(w).lib version linked" – invalid medqb.lib or medqbw.lib library was linked into the application. Please check your link file.

10 - "Unable to set transaction mode" – application was unable to set correct transaction mode on login.

11 - "Invalid OMVDD.DLL version"- application cannot connect to Mediator server because of the invalid version of OMVDD.DLL installed on the workstation.

15 - "Cancelled at user request"- user has cancelled the login process

16 -"Medconed.exe failed to start"- (x)Harbour application was unable to run ***medconed.exe*** connection editor

17 – “Unable to login - invalid application/global password” – application failed to supply valid global application password defined in Mediator server. Application password should be attached at the end of user password in square brackets (ie. userpasswd[apppasswd]).

18 – “Server maintenance – connection forbidden” – maintenance mode has been activated on Mediator server and connection cannot be established. Contact Mediator server administrator.

19 – “Communication error” – error in client-server communication

20 – error reported by the database server – the text and code of an error can be acquired by calling **MedCmdRes** and **MedErrText** functions right after **MedLogin** function.

21 - "Unable to login - no OS user defined in Mediator server" – operating system authentication is selected in Mediator server but the given user is not defined in Mediator as operating system user and no mapping to database user is specified.

22 – error reported by operating system – the text of an error can be acquired by calling **MedErrText** functions right after **MedLogin** function.

23 – ”Enterprise Mediator server required for this client application” – this application requires Enterprise version of Mediator server to connect.

24 – “Too many global users (Lock Manager)” – number of Mediator users working with all Mediator servers supervised by Lock Manager module over-passed the license limit.

25 – “End of server evaluation period or invalid system date setting” – for evaluation versions of Mediator server - evaluation period has finished or system date is not correctly set on the server and/or client machines.

26 – “Mediator server reports too many Terminal server users” – Mediator server has detected that the number of connections originating from terminal applications exceed the licence limit of the cooperating Terminal server.

27 – „Invalid charset specified”– Invalid value for *cCharset* parameter was specified. For instance CHARSET_UNICODE was specified by CA-Clipper application.

Error message describing the reason of login failure can be displayed by calling MedLgMsg() function with the error code returned by MedLogin(). MedLogErr() function can be used to raise error with appropriate message.

Example:

```
logRes = MedLogin(
”00000123”,”000000000001”,”4546”,”test”,”password”,””)
IF logRes != 0
   ? ”Unsuccessful connection attempt!”
   ? ”Error: ”, MedLgMsg(logRes)
QUIT
```

```
ENDIF
? "Connection OK!"
...
MedLogout()
```

See also: **MedRegLogin(), MedLogErr(), MedLgMsg()**

92. MedLogout

Syntax:

```
MedLogout()
```

The function closes an existing connection to the Mediator server. All working areas are closed, changes are saved to the database and if a transaction is being processed, it is rolled back.

93. MedLogTVal

Syntax:

```
MedLogTVal(<nTrueValue>) -> nPreviousTVal
```

If logical fields are implemented as NUMBER(2) (see *MedLogAsN2()* function), default values used for logical fields are

0 – false (.F.)

1 – true (.T.)

Using *MedLogTVal*() function you can redefine value to be used as *True* (.T.). New value should be integer in the range –9 to 99. Calling *MedLogTVal(-1)* at the beginning of your program will instruct Mediator server to use –1 when this session inserts logical .T. value into record.

Please note, that all values but 0 are interpreted by Mediator server as *True* (.T.)

See also: **MedLogAsN2(), MedMemType(), MedColAdd()**

94. MedMaxLic

Syntax:

```
MedMaxLic() -> nMaximumLicences
```

The function returns the number of licenses (connections) supported by Mediator server.

See also: **MedDbConn()**

95. MedMedId

Syntax:

```
MedMedId() -> nMediatorId
```

The function returns the numeric ID of the Mediator server within the farm of Mediator servers managed by Lock Manager module. If Mediator server works without Lock Manager supervision its ID is -1 and the function returns -1. Before using *MedMedId()* function, application should be connected to Mediator server.

See also: ***MedClntId, MedRLckInf, MedFLckInf, MedSetInfo, MedGetInfo***

96. MedMedVBFx

Syntax:

```
MedMedVBFx() -> nMediatorBugFix
```

The function returns the integer number that represents the small patch number of the Mediator server.

97. MedMedVMaj

Syntax:

```
MedMedVMaj() -> nMediatorMajorVersion
```

The function returns the integer number that represents the main version number of the Mediator server.

98. MedMedVMin

Syntax:

```
MedMedVMin() -> nMediatorMinorVersion
```

The function returns the integer number that represents the detailed version number of the Mediator server.

99. MedMedVPth

Syntax:

```
MedMedVPth() -> nMediatorPatchVersionASCII
```

The function returns the ASCII code of the letter representing Mediator server version modification. This can be a code of ' ' (space) for base version or a code of letter 'a' to 'h' representing the modification level.

100. MedMedVSub

Syntax:

```
MedMedVSub() -> nMediatorSubVersion
```

The function returns the integer number that represents the patch number of the Mediator server.

101. MedMemType

Syntax:

```
MedMemType( nMemoType ) -> nPrevMemoType
```

The function specifies the type of fields where MEMO values are to be stored. It also allows to specify if MEMO values should be stored in main table or auxiliary table (TABLENAME_MEMO). After calling the **MedMemType** function, all tables created with the "MEDNTX" (or "MEDCDX") driver will use that type for the MEMO fields implementation. The default type is binary type (*MED_MEMO_LONGRAW*), which allows to store big values (for Oracle it is LONG ROW). The function argument can be one of constants defined in *mediator.ch* file:

MED_MEMO_LONG **- MEMO stored in auxiliary table as text type (Oracle–LONG, MS SQL-TEXT)**

MED_MEMO_LONGRAW **- MEMO stored in auxiliary table as binary type (Oracle-LONG RAW, MS SQL-IMAGE)**

MED_MEMO_BLOB **- MEMO stored in main table as large binary type (Oracle-BLOB, MS SQL-IMAGE)**

MED_MEMO_CLOB **- MEMO stored in main table as large character object (Oracle-CLOB,MS SQL-TEXT)**

When implementing MEMO fields as binary (*MED_MEMO_LONGRAW, MED_MEMO_BLOB*), the conversion of character codes is not executed according to SET CLIENT CODE PAGE and SET SERVER CODE PAGE or xHarbour codepage settings. The lack of conversion allows storage of binary data in MEMO fields. If MEMO field is defined as text type the code page conversion is active.

The function should be called before the table creation or using *MedColAdd()* function. After the table is created its MEMO field type cannot be changed.

See also: ***MedLogAsN2(),*** **SqlMemType()**

102. MedMrkAdd

Syntax:

MedMrkAdd(<*nHandle*>, <*nRecno*>)

The function adds the record with recno equal to *nRecno*, to the marker table identified by *nHandle*, and optionally with the value of the field specified during its opening. Subsequent markers inserted by **MedMrkAdd** are buffered in the workstation until the buffer overflow or calling one of the functions: **MedMrkClose** or **MedMrkFlush**.

Example:

```
mhdl = MedMrkOpen( ”mark01”, ”goods”, ”good_id”)
IF mhdl == 0
   return
ENDIF
MedMrkAdd( mhdl, 1 )
MedMrkClose( mhld )
```

103. MedMrkAll

Syntax:

```
MedMrkAll( <nHandle> )
```

The function inserts the records for all recno values existing in *cMarkTable* and optionally the fields from the *cMarkTable* table (see: **MedMrkNew**) to the marker table identified by *nHandle*. The function operation is not buffered. If a DELETED parameter is turned off (SET DELETED ON), only these records that are not deleted are taken into consideration.

Example:

```
mhdl = MedMrkOpen( ”mark01”, ”goods”, ”good_id”)
IF mhdl == 0
   return
ENDIF
MedMrkAll( mhdl  )
MedMrkClose( mhld )
```

104. MedMrkClose

Syntax:

```
MedMrkClose( <nHandle> )
```

The function closes the marker table and frees *nHandle* identifier. Before closing all markers that are buffered on the client and added or deleted by **MedMrkAdd** and **MedMrkDel** are sent to database.

Example:

```
mhdl = MedMrkOpen( ”mark01”, ”goods”, ”good_id”)
IF mhdl == 0
   return
ENDIF
MedMrkAdd( mhdl, 1  )
MedMrkClose( mhld )
USE tab AS ‘select sum(amount) from goods t where
exists ( select recno from mark01 where recno =
t.recno )’
....
USE
```

105. MedMrkDel

Syntax:

```
MedMrkDel( <nHandle>, <nRecno> )
```

The function deletes from mark table the record with recno equal to *nRecno* and optionally with the value of a record specified while opening. Subsequent markers that are deleted by **MedMrkDel** are buffered on the workstation until the buffer overflow or calling one of the functions: **MedMrkClose**, **MedMrkFlush.**

Example:

```
mhdl = MedMrkOpen( ”mark01”, ”goods”, ”good_id”)
IF mhdl == 0
   return
ENDIF
MedMrkDel( mhdl, 1 )
MedMrkClose( mhld )
```

106. MedMrkFlush

Syntax:

```
MedMrkFlush( <nHandle> )
```

The function immediately sends to server all markers buffered on the client and added or deleted by **MedMrkAdd** and **MedMrkDel** functions. This function allows the use of a marker table without closing the marker.

Example:

```
mhdl = MedMrkOpen( ”mark01”, ”goods”, ”good_id”)
IF mhdl == 0
   return
ENDIF
MedMrkAdd( mhdl, 1  )
MedMrkFlush( mhld )
USE tab AS ‘select sum(amount) from goods t where
exists ( select recno from mark01 where recno =
t.recno )’
....
USE
```

107. MedMrkNew

Syntax:

```
MedMrkNew(<cMarkName>, <cMarkedTable>
[,<cMarkedField>]) -> nHandle
```

This function creates an empty marker table named *cMarkTable* in a database. The *cMarkTable* parameter can contain a name of a RDBMS user who is to be the owner of that table (a current database user by default). This table will contain RECNO numbers of records from the *cMarkedTable* table (it is possible to specify the name of the user who is the table owner). If the *cMarkedField* name from the *cMarkedTable* table is given, then in addition to RECNO the marker table contains the *cMarkedField* column with values copied from the *cMarkedTable*. The function returns a positive number (the handle) that identifies a marker table in further operations. In case of an error, the function returns 0. If a table named *cMarkName* exists in RDBMS, the function will return an error.

Example:

```
mhdl = MedMrkNew( ”mark01”, ”goods”, ”good_id”)
IF mhdl == 0
   return
ENDIF
MedMrkAll( mhdl )
MedMrkClose( mhld )
```

108. MedMrkNum

Syntax:

```
MedMrkNum( <nMaxMarkNum> )
```

The function specifies the maximum number of simultaneously opened markers (default is 5). It needs to be called before the first call of **MedMrkNew** or **MedMrkOpen** function.

Example:

```
MedMrkNum(20)
mhdl = MedMrkOpen( ”mark01”, ”goods”, ”good_id”)
```

109. MedMrkOpen

Syntax:

```
MedMrkOpen(<cMarkName>, <cMarkedTable>
[,<cMarkedField>]) -> nHandle
```

The function opens an existing marker table named *cMarkTable*. The *cMarkTable* parameter can contain the RDBMS name of the user who is the owner of that table. The *cMarkedTable* and *cMarkedField* arguments should have identical values with the ones used during the creation of marker table (**MedMrkNew** function call). The function returns a positive handle number that identifies the marker table in further operations. In case of an error, the function returns 0. If a table named *cMarkName* does not exist in RDBMS, the function will return an error.

Example:

```
mhdl = MedMrkOpen( ”mark01”, ”goods”, ”good_id”)
IF mhdl == 0
   return
ENDIF
MedUMrkAll( mhdl )
MedMrkClose( mhld )
```

110. MedMrkRemv

Syntax:

```
MedMrkRemv( <nHandle> | <cMarkName> )
```

The function deletes the marker table identified by *nHandle* or *cMarkName* name. If the table *cMarkName* does not exist in RDBMS, the function will not return an error.

Example:

```
mhdl = MedMrkOpen( ”mark01”, ”goods”, ”good_id”)
IF mhdl == 0
   return
ENDIF
MedMrkAdd( mhdl, 1  )
....
MedMrkRemv( mhld )
```

111. MedMrkTemp

Syntax:

```
MedMrkTemp( [<lUseTempMarks>] ) -> lPreviousSettings
```

Calling *MedMrkTemp(.T.)* instructs Mediator server to preferably use temporary database objects when creating mark tables. Operations performed on temporary objects are usually faster but these objects are automatically deleted when the current database session is closed.
Calling *MedMrkTemp(.F.)* restores the standard mode for mark tables creation. Function returns previous setting of the flag controlling mark table creation mode.

112. MedNoGoTop

Syntax:

```
MedNoGoTop( <lNoGoTopSetting> ) -> lPreviousSetting
```

Use *MedNoGoTop()* function to temporarily block sending of dbGoTop()/GO TOP commands to Mediator Server. Many commands and functions such as dbUseArea(), dbSetIndex(), dbClearIndex(), MedSetOFil() execute dbGoTop() to correctly position current table record. In practice, it often results in many unnecessary executions of dbGoTop(). To optimize your application and prevent unnecessary dbGoTop() executions, call *MedNoGoTop(.T.)* which blocks all *dbGoTop()* commands sent to the server. Remember to unblock dbGoTop() command by calling *MedNoGoTop(.F.)* and to execute dbGoTop() yourself when required. When dbGoTop() command is blocked, the table seems to be empty (EOF() .AND. BOF()).

Example:

```
MedNoGoTop(.T.)      && block GOTOPs
USE table SHARED VIA "MEDNTX"
SET INDEX TO INDEX1
SET SQL FILTER TO "SALARY <= 10000"
MedNoGoTop(.F.)      && unblock GOTOPs
dbGoTop()      && one GOTOP instead of three
```

113. MedNulChar

Syntax:

```
MedNulChar() -> cRDBMSNullCharacter
```

If client is connected to the database, the function returns the character, which is used in the database for empty character fields. If there is no connection to the database, null string is returned. The purpose of function is using it in SQL queries and filters.

Example:

```
USE qry AS ”SELECT count(*) FROM tab WHERE cfield <>”
+ MedNulChar()
```

114. MedNulDate

Syntax:

```
MedNulDate() -> cRDBMSNulldate
```

If the client is connected to the database, the function returns string representing the date used in the database for empty date fields. If there is no connection to the database, null string is returned. The purpose of function is using it in SQL queries and filters.

Example:

```
SET SQL FILTER TO "date_field <>” + MedNuldate()
```

115. MedOci8

Syntax:

```
MedOci8() -> lOracleMediatorOCI8orNewer
```

This function returns .T. if working with Mediator for Oracle using OCI interface 8 or newer. Otherwise function returns .F.

116. MedOpSpeed

Syntax:

```
MedOpSpeed([<nOpSpeed>]) -> nPrevOpSpeed
```

Function limits the speed of the application accessing Mediator Server. *nOpSpeed* parameter specifies the maximum allowed speed in operations per second. Setting speed to 0 (zero) turns the limitation off. If the used client-server configuration is slower then the specified limit the limit does not work.

Executing on workstation some „havy” operations such as reports place a high load on server or network. MedOpSpeed function allows you to limit the resources taken by a single workstation. Using this function you can enforce the limit during reports e.t.c.

The following operations are taken into account while limiting the speed:
- SKIP
- SEEK
- GOTO
- GO TOP
- GO BOTTOM
- RLOCK/FLOCK
- APPEND

117. MedPdbFree

Syntax:

`MedPdbFree()`

Function removes from application memory all parameters loaded using *MedPdbLd()* function.

See also: ***MedPdbLd(), MedPdbGet(), MedPdbSet()***

118. MedPdbGet

Syntax:

`MedPdbGet(<cSection>,<cParamName>) -> cParamValue`

Function return value of the parametr *cParamName* defined in function *cSection*. If such parameter does not exist, function returns empty string.

See also: ***MedPdbLd(), MedPdbFree(), MedPdbGet(), MedPdbSet()***

119. MedPdbLd

Syntax:

MedPdbLd(<*cParamFileName*>) -> *nParamLoaded*

Function is used to load parameters defined in *cParamFileName* file into application memory. Loaded parameters can be accessed using *MedPdbGet()* function. Use *MedPdbFree()* function to remove parameters and free memory. Function returns the number of successfully loaded parameters. Zero (0) return value means there are no correctly defined parameters in *cParamFileName* file or file does not exist.

Every parameter in *cParamFIleName* should be defined using the following syntax

[SECTION:]PARAMETER=PARAMETER VALUE

SECTION is optional, PARAMETER defines parameter name.
PARAMETER VALUE is everything after '=' character till the end of the line. Parameter and section name are not case sensitive. Lines starting with '#' character are ignored and can be used as comments.

Successive calls to *MedPdbLd()* function result in appending new parameters to existing ones. To remove all parameters from application memory call *MedPdbFree()*

Example

```
nPar = MedPdbLd(”medapp.ini”)
IF nPar == 0
   ? ”No parameters loaded!”
   QUIT
ENDIF
node = MedPdbGet(””,”mednodeaddr”)
socket = MedPdbGet(””,”medsocket”)
user = MedPdbGet(””,”meduser”)
password = MedPdbGet(””,”medpasswd”)
cs = MedPdbGet(””,”medcs”)
MedPdbFree() && clear loaded parameters
res = MedLogin(””,node,socket,user,passwd,cs)
```

```
...
MedLogout()
```

See also: ***MedPdbGet(), MedPdbSet(), MedPdbFree()***

120. MedPdbSet

Syntax:

```
MedPdbSet(<cSection>,<cParamName>,<cValue>) ->
lSuccess
```

Function adds new parameter to parameter database kept in application memory. If parameter *cParamName* in *cSection* section already exists its value is replaced with *cValue*. If such parameter dos not exist – it is added to parameter database.

Function returns .T. if succesfull, .F. if parameter cannot be added. Parameter database can hold up to 256 parameters.

See also: ***MedPdbLd(), MedPdbFree(), MedPdbGet()***

121. MedPerfMod

Syntax:

```
MedPerfMod() -> lTableMode
```

This function returns logical value indicating the open mode of the table, opened in the current workarea.

Result *.T.* means that the table was opened (or later converted) in a non-continuous record numbering mode (with possibly perforated RECNO).

Result *.F.* means that the table was opened in a continuous record numbering mode (this is 100% CA-Clipper compatible mode)

See also: **SET PERFORATED NUMBERING, MedSetPerf(), MedPerfRC()**

122. MedPerfRC

Syntax:

```
MedPerfRC() -> nReccount
```

This function returns the number of records present in the table, opened in the current workarea. As opposed to RECCOUNT() and LASTREC() functions *MedPerfRC()* behaves correctly on tables with non-continuous (perforated) record numbering. In such a case RECCOUNT() and LASTREC() functions return the number of the last record, which may be different from the actual number of records present in the table.

Due to the high server load generated, MedPerfRC() function should be used only in situations where it is really necessary. In most cases it is possible to use RECCOUNT() or LASTREC() function instead.

See also: **SET PERFORATED NUMBERING, MedSetPerf(), MedPerfMod()**

123. MedPrepStm

Syntax:

```
MedPrepStm( <cSQL> [,<nPrecision>] ) -> nStmtHandle
```

Function prepares *cSQL* SQL statement for multiple execution by *MedExecStm()*. SQL statement can be of arbitrary type but for SELECT statement MedExecStm() will return just one value, exactly like *MedSelVal()* does. Function returns the prepared statement handle. This handle can be used when calling *MedExecStm()* and when execution finished, should be released by passing it to *MedFreeStm()* function.. Using *MedPrepStm()* -> *MedExecStm()* -> *MedFreeStm()* call seqence is meaningfull only if you wish to execute the statement many times with different parameter sets. In such a case we reduce the time spent by database server parsing the statement and binding the parameters. In practice, the acceleration depends on the database server involved and type of SQL statement. Generally, the faster statement, the better optimization. Thus, best results can be expected for SELECT statements.

In case of SELECT returning numerical value being result of expression (and not of the field in the table), that value will be rounded according to default precision specified with SET QUERY PRECISION command, or precision *nPrecision* specified as optional parameter which is the number of decimal places required in the result.

For explanation how parameters can be specified for SQL statements, please see description of *MedSqlPar()* function.

Example:

```
MedSqlPar(1) && set example parameter
hStmt = MedPrepStm("SELECT salary FROM tab WHERE eid=:1")

FOR id = 1 to 1000
   MedSqlPar(id)&& set param for next execution
   ? "Employee",id,"earns", MedExecStm(hStmt)
NEXT
MedFreeStm(hStmt)      && free allocated statement
```

See also: **MedExecStm(), MedFreeStm(), MedSelVal()**

124. MedRddUser

Syntax:

```
MedRddUser() -> cRddUserName
```

The function returns the MEDIATOR user name (the user's ID that has been specified during the program execution or read from DOS environment).

125. MedRegLogin

Syntax:

```
MedRegLogin(<cConnectionName>) -> nResult
```

It enables the creation of the connection between the client and the Mediator server/RDBMS.

The function input parameter is the name of the connection defined in the Windows registry in the subtree HKEY_LOCAL_MACHINE/SOFTWARE/OTC/MEDIATOR/CONNECTIONS.

The function reads arguments required to establish the connections from the keys in the Windows registry and calls the *MedLogin()* function. To establish the connections, the following keys should be defined in the given node: MEDNETADDR, MEDNODEADDR, MEDSOCKET, MEDUSER, MEDPASSWD, MEDCS.

More information on the *MedLogin()* function can be found in the manual pages of that function. The return values of *MedRegLogin()* are identical as those of *MedLogin().*

Example

```
logRes = MedRegLogin("Default")
IF logRes != 0
   ? "Connection could not be established!"
   QUIT
ENDIF
? "Connection OK!"
...
MedLogout()
```

See also: **MedLogin()**

126. MedRenTab

Syntax:

MedRenTab(<*cOldName*>, <*cNewName*>)

This command renames the table *cOldName* to *cNewName*. The table for renaming operation is locked, so before the execution of **MedRenTab** command the table must not be used by another user. If the table is currently in use, an error is generated. The function can be executed only by the owner of the *cOldName* table.

Example:

```
MedRenTab("old_tab",  "new_tab")
```

This command renames the table named **old_tab** to **new_tab**.

127. MedRfsRecA

Syntax:

MedRfsRecA([<*nRefreshFlags*>]) -> *nPreviousFlags*

Function sets global flags controlling automatic refresh of the maximum RECNO value performed by Mediator server before executing some commands. Refresh is performed by reading the maximum RECNO from database table just before the main operation/command is executed.

Calling *MedRfsRecA(F_RECNORFSH_SEEK*) will instruct Mediator server to read maximum RECNO value from table each time before performing SEEK operation. *MedRfsRecA()* function sets global flags controlling all tables used by application. Use *MedRfsRecT()* function to selectively set refresh flags for a given table.

Auto-refreshing maximum RECNO value can be useful when non-Mediator applications insert records to tables opened via "MEDNTX" or "MEDCDX" driver. Refreshing RECNO before operation, ensures the operation will be aware about new externally inserted records.

Turning on automatic refresh of maximum RECNO value for all tables may have serious performance impact on the application. Therefore, in most cases it is preferable to turn on automatic refresh only for tables where it is important. It can be done using *MedRfsRecT()* function.

MedRfsRecA() function can be called with the following parameters (flags defined in Mediator.ch file):

0 – turning off automatic RECNO refresh for all operations

F_RECNORFSH_SEEK – turning on automatic refresh before SEEK command

...

Function returns previous global refresh flag settings.

See also: ***MedExRecc(), MedRfsRecc(), MedRfsRecT()***

128. MedRfsRecc

Syntax:

`MedRfsRecc( [ <nWorkareaNumber> ] )`

Function forces Mediator Server to refresh its internal record count for current (or specified as *nWorkareaNumber*) workarea. This function can be useful when non-Mediator applications are appending records to tables opened via "MEDNTX" ("MEDCDX") driver. Such "externally" appended records will not be seen by RECCOUNT() and LASTREC() functions until table is reopened or *MedRfsRecc()* function is called.

See also: **MedExRecc()**

129. MedRfsRecT

Syntax:

MedRfsRecT([<*nTabRefreshFlags*>]) -> *nPreviousFlags*

Function sets flags controlling automatic refresh of the maximum RECNO value for table opened in current workarea. Refresh is performed by reading the maximum RECNO from a given table just before the main operation/command is executed.

Calling *MedRfsRecT(F_RECNORFSH_SEEK*) will instruct Mediator server to read maximum RECNO value from a given table each time before performing SEEK operation. Selective flags set for a given table using *MedRfsRecT()* function have priority over global flags set with *MedRfsRecA()* function.

Auto-refreshing maximum RECNO value can be useful when non-Mediator applications insert records to tables opened via "MEDNTX" or "MEDCDX" driver. Refreshing RECNO before operation, ensures the operation will be aware about new externally inserted records.

Function parameters are identical to that of *MedRfsRecA()* function.

Function returns previous refresh flag settings for table opened in current workarea.

See also: ***MedExRecc(), MedRfsRecc(), MedRfsRecA()***

130. MedRlckInf

Syntax:

```
MedRlckInf( <nRecNo>, [@<nMedSessId>],
[@<nMediatorId>], [@<nLockType>], [@<dLockDate>],
[@<cLockTime>] ) -> lSuccess
```

The function returns diagnostic information related to the last unsuccessful attempt to lock record *nRecNo* in the current workarea. It can be called only right after unsuccessful attempt to lock the record. Function returns *.T.* if diagnostic info was successfully read or *.F.* otherwise. When successful, function assigns the following information to the passed by reference function parameters:

nMedSessId – number of the Mediator session which placed the existing lock making it impossible to place the new lock. Each application can obtain its session number using *MedClntId()* function.

nMediatorId – number of the Mediator server connected with the session which placed the existing lock making it impossible to place the new lock. Each application can obtain its Mediator number using *MedMedId()* function. (*nMedSessId*, *nMediatorId*) pair uniquely identifies application working with Mediator server within the farm managed by Lock Manager module.

nLockType – type of the existing lock: MED_LOCK_RLOCK (1), MED_LOCK_FLOCK (2) or MED_LOCK_APPEND (4). MED_LOCK_APPEND means the existing lock has been automatically placed by dbAppend() function.

dLockDate – date (day) when the existing lock was placed

cLockTime – time when the existing lock was placed as a string in the format "HH:MM:SS", 24-hour mode.

Information returned by *MedRlckInf()* function can be used with *MedGetInfo()* function to read the extra information set by the application which placed the existing lock. Alternatively, application can use its own tables containing *nMedSessId* and *nMediatorId* of every application to find the application user who placed the conflicting lock.

See also: ***MedFlckInf, MedSetInfo, MedGetInfo***

131. MedSelStrVal

Syntax:

```
MedSelStrVal( <cSQLSelect> [,<nResultEncoding>] ) ->
cValue
```

xHarbour/Harbour only. The function allows easy reading of the single character values from the database. It differs from *MedSelVal()* function in that it is limited to character values only but allows the specification of the returned value encoding format.

Parameter *nResultEncoding* specifies the encoding for the returned character value. It can be one the following:

STRENC_DFLT – depending on the connection type the value is read in default database UNICODE encoding (CHARSET_UNICODE connection) or default database single-byte codepage (CHARSET_SB connection). Than it is returned to the application after being converted to a single-byte codepage set using HB_SETCODEPAGE() function.

STRENC_SB - string is read from the database using single-byte database client codepage (for instance EE8MSWIN1250 for Oracle/Win), sent to RDD driver and returned to the application with no other conversions.

STRENC_UTF8 - string is read from database as UTF8, send to RDD driver and returned to the application without conversion as a string value containing a sequence of UTF8 bytes.
STRENC_UTF16 - string is read from database as UTF16, send to RDD driver and returned to the application without conversion as a string value containing a sequence of UTF16 bytes.
STRENC_UCS2 - string is read from database as UCS2, send to RDD driver and returned to the application without conversion as a string value containing a sequence of UCS2 bytes.

Not every encoding value is valid for every database. For Oracle every described encoding is possible. For MS SQL Server only STRENC_SB and STRENC_UCS2 encodings are valid.

The default value for *nResultEncoding* parameter is STRENC_DFLT.
For character values the MedSelVal() function is the equivalent of the MedSelStrVal() function called with STRENC_DLFT encoding format.

Example:

```
MedSqlPar(100)

val = MedSelStrVal("SELECT description FROM tab WHERE
id=:1", STRENC_UTF8)

IF val == NIL
   ? "Description not found"
ELSE
   ? "We have now UTF8 string in val"
ENDIF
```

See also: **MedSelVal(), USE AS, MedSqlPar(), MedSQLParA()**

132. MedSelVal

Syntax:

```
MedSelVal( <cSQLSelect> [,<nPrecision>] ) -> value
```

The function allows reading single values from the database easily. The *cSQLSelect* argument specifying SELECT command for execution on the database server should be formed in such way, that only a single value is returned. That value will be returned as result of the function. If SELECT command specified returns more than one row, the function will return the value from the first row read. If SELECT does not return any records, NIL value will be returned.

In case of SELECT returning numerical value being result of expression (and not of the field in the table), that value will be rounded according to default precision specified with SET QUERY PRECISION command, or precision *nPrecision* specified as optional parameter which is the number of decimal places required in the result.

For explanation how parameters can be specified for SQL statements, please see description of *MedSqlPar()* function.

Example:

```
MedSqlPar(50)

val = MedSelVal("SELECT salary FROM tab WHERE eid=:1")
IF val == NIL
   ? "Employee 50 not found"
ELSE
   ? "Employee 50 earns ", val
ENDIF
```

See also: **MedSelStrVal(), USE AS, SET QUERY PRECISION, MedSqlPar(), MedExecSQL()**

133. MedSetDefColEnc

Syntax:

```
MedSetDefColEnc( nDefColumnEncoding ) -> nPrevSettings
```

Function available for Harbour/xHarbour only. Allows the setting of the default database encoding type for character columns. Database must support the specified encoding. This setting is used by dbCreate (and similar) function and MedColAdd() function to decide which database type to use for character column.

nDefColumnEncoding parameter can be one the following:

- STRENC_DFLT – default setting – character columns will be created to store values determined by the type of the connection established between the application and Mediator server (see MedLogin()). For CHARSET_SB connection these will be single-byte character values, for CHARSET_UNICODE connection these will be UINCODE values encoded using the database default UNICODE encoding.
- STRENC_UTF8 – character columns will store UNICODE values encoded as UTF8
- STRENC_UTF16 – character columns will store UNICODE values encoded as UTF16
- STRENC_UCS2 – character columns will store UNICODE values encoded as UCS2

Depending on the database type and configuration not all of the possible variants are allowed. If you specify an invalid encoding for a given database/configuration the "The requested characterset encoding cannot be set" error will be raised. The function returns the previous parameter setting.

See also: ***MedLogin(), MedColAdd()***

134. MedSetSqlEnc

Syntax:

```
MedSetSqlEnc( nSQLEncoding ) -> nPrevSettings
```

Function allows to specify the text encoding for character values which are not destined for database character fields. It is used for the following strings:

- SQL command text set by MedExecSQL(),MedSelVal(), MedSelStrVal(), MedSetQry(), USE AS SELECT
- index expressions created and read via MEDNTX/MEDCDX RDD driver
- client and SQL filters set by the application

nSQLEncoding parameter can be one of the following:

- STRENC_SB – character values will be sent to Mediator using single-byte encoding.
- STRENC_UTF8 – character values will be sent to Mediator as UNICODE values encoded as UTF8

- STRENC_UTF16 – character values will be sent to Mediator as UNICODE values encoded as UTF16
- STRENC_UCS2 – character values will be sent to Mediator as UNICODE values encoded as UCS2

Depending on the database type and configuration not all of the possible encodings are possible. If you specify an invalid encoding for a given database/configuration an error will be raised. The function returns the previous parameter setting

See also: **MedLogin()**

135. MedSessId

Syntax:

```
MedSessId() -> nOracleSessionId
```

For Oracle - this function returns the Oracle session number opened for this client. For other servers it returns Mediator session number.

136. MedSetInfo

Syntax:

```
MedSetInfo( [<cName32>], [<cString32>],
[<cString128>], [<nInt1>], [<nInt2>], [<nInt3>] )
```

The function allows to define a set of parameters related to the current Mediator session (application). Each application can define its own set of parameter values which can be read by other applications working with the same Mediator server or other Mediator server within the same farm managed by the Lock Manager module. Application can set all the parameters or only some of them. If a given parameter is missed on the function parameters list its value stored in Mediator server will remain unchanged. Parameters set using *MedSetInfo()* function can be read by any application using *MedGetInfo()* function. These parameters have no defined semantics (meaning). It is up to the application programmer to define their meaning within the application. Parameter values are also accessible via mmt.exe tool. They can be inspected within session details section. cName32 parameter is also visible in mmt.exe on the session list.

cName32 – max 32-character string
cString32 – max 32- character string

cString128 – max 128- character string
nInt1 – max 32-bit integer value
nInt2 – max 32-bit integer value
nInt3 – max 32-bit integer value

Used together with *MedRlckInf()* and *MedFlckInf()* functions, session parameters can be effectively used for providing detailed info about the locking problems or deadlock situation.

See also: ***MedGetInfo, MedRlckInf, MedFlckInf***

137. MedSetPerf

Syntax:

`MedSetPerf(<lNewMode>) -> lPreviousMode`

This function changes the mode used when opening tables. After the calling of*MedSetPerf(.T.)* all following tables are opened in the mode allowing a non-continuous record numbering (perforated RECNO mode). After the calling of *MedSetPerf(.F.)* all following table openings are made in the CA-Clipper compatible mode, which does not allow a non-continuous record numbering. On an application startup, the CA-Clipper compatible mode is active (no perforated RECNO allowed).

The function returns the logical value indicating the previously active open mode. If called without the parameters, the function returns the active mode but does not change it.

Warning!
If at least one application opens the table in a perforated numbering mode, the opening mode for this table is automatically converted to “perforated” for all other applications, including the applications having this table already opened. In such a situation, if another application tries to open the table in a “non-perforated” mode, the “perforated” opening will be forced. All that means that the table opening mode may change from a “non-perforated” to a “perforated”, even after the table is opened. The vice-versa change change is not possible. To check the actual mode of the table use *MedPerfMod()* function.

To avoid confusion, related to numbering modes, OTC strongly recommends opening a given table from all applications in the same numbering mode.

However, it is perfectly legal (and very useful) to have in one application tables opened in “perforated” and “non-perforated” modes. The typical candidates for “perforated” opening are tables to which you add records within the transaction.

Opening tables in a non-continuous numbering (perforated) mode has been possible since 3.0 version of Mediator. When the table is opened in a non-continuous record numbering mode, then Mediator does not need to wait to append a new records to the table until all other transactions inserting into this table are finished, but immediately calculates the new record number and proceeds with its insertion. This method may cause the “holes” in a record numbering. Such a “numbering hole” may be created if record inserting transaction is rolled back at the moment when other applications have already inserted records with higher record numbers. The records inserted within the rolled back transaction are removed, and a “numbering hole” remains. Using a non-continuous (“perforated”) numbering mode facilitates the usage of transactions (eliminates the possibility of the deadlock), but has the following consequences:

- RECCOUNT() and LASTREC() functions called on the table with non-continuous record numbers return the number of the last record, not the actual number of records. To obtain the accurate number of records in such a table, call *MedPerfRC()* function.
- The application may not assume a continuous record numbering
- An attempt to GOTO to the record which does not exist, results in positioning the cursor on the phantom record (EOF()==.T.).

See also: **SET PERFORATED NUMBERING, MedPerfMod(), MedPerfRC()**

138. MedSetScpe

Syntax:

`MedSetScpe(<nScope> [,<xValue>]) -> currentValue`

Attention: in Harbour and xHarbour versions of Mediator client you can use standard OrdScope() function to manipulate scopes.

The function corresponds to the ORDSCOPE() function from the CA-Clipper library. It allows specifying scope by setting minimum and maximum index key of table records. After setting, the logical beginning of the table is record of minimum allowed value of index key, and the logical end of the table is the record with

maximum allowed value of the key. The *nScope* argument specifies the limit. Values allowed:

0 – the low limit (top scope)

1 – the high limit (bottom scope)

The *xValue* argument specifies value of the limit set. The value is to be compliant with the type of index active in current workarea. Specifying NIL as the new limit deletes the limit. The function returns current value of the limit.

Scopes can be set on indexes of tables opened with MEDNTX or MEDCDX driver.

Example:

```
USE friends VIA "MEDCDX"
SET INDEX TO age

// 25 becomes the lowest age in range
MedSetScpe(0,25)
GO TOP
BROWSE()

// 30 becomes the lowest age in range
MedSetScpe(1,30)
GO TOP
BROWSE()

// change lowest age to 20
MedSetScpe(0,20)
GO TOP
BROWSE()

// Clear top boundary
MedSetScpe(0,NIL)
GO TOP
BROWSE()

// Clear bottom boundary
MedSetScpe(1,NIL)
GO TOP
BROWSE()
```

See also: **MedClrScpe(), CA-Clipper documentation: ORDSCOPE()**

139. MedShared

Syntax:

```
MedShared() -> lResult
```

Function returns .T. if table opened in current workarea is opened in shared (SHARED) mode. Otherwise function returns .F.

See also: **MedFlocked()**

140. MedSqlPar

Syntax:

```
MedSqlPar( <SQLpar1>, ...)
```

Function defines parameter values for SQL statements. SQL statements can be executed using MedSelVal(), MedExecSQL() ,USE qry AS 'select ..', USE qry AS PROC/FUN functions/commands. SQL statements passed to the above functions may contain parameters specified as ':1', ':2' till ':99'. To specify values to be used in place of these parameters you need to call *MedSqlPar()* function just before executing any of the above SQL commands. You specify values to be used in SQL statement as parameters to *MedSqlPar()* function. First *MedSqlPar()* function parameter corresponds to ':1' parameter in SQL statement and so on.

Calling any of the above SQL functions/commands automatically clears the parameter values set by MedSqlPar() call, so before next SQL statement you need to call MedSqlPar() function again specifying new parameter values.

For all database servers parameters are specified in SQL statement using the same syntax, namely ':n'.

ATTENTION!

Parameter values are sent to Mediator Server and binded to SQL statement on the server. It means repeating execution of the same SQL statement with different parameter values can be effectively optimized by database Server. That's why using parameters is preferred to embedding their values directly in SQL statement.

Example

```
Indeks = 1000

MedSqlPar(indeks,20,'John')
```

```
val = MedSelVal('SELECT sum(profit) FROM tab WHERE
emp=:3 and itemid=:1 and itemprice>:2')

IF val == NIL
   ? "No such profits"
ELSE
   ? "Calculated profit:", val
ENDIF

&& now remove all records produced by John

MedSqlPar('John')

MedExecSql('DELETE FROM tab WHERE emp=:1')

&& now browse all records related to item Indeks

MedSqlPar(Indeks)

USE qry AS 'SELECT * FROM tab WHERE itemid=:1'
SCROLLABLE
BROWSE()
```

See also: **USE AS, SET QUERY PRECISION, MedExecSQL(), MedSqlParEx(), MedSelVal(), MedSqlPTrm()**

141. MedSqlParA

Syntax:

MedSqlParA(<a*SQLparams*> [,<anCharParEncoding>].)

Function can be used to define SQL expression parameters. It is identical to MedSqlPar() but parameters are passed as Clipper/Harbour array instead of individual values.

In xHarbour/Harbour applications you can pass a second, optional *anCharParEncoding* parameter. It is an array which specifies the encoding used for the specified character parameters. This table should be of the same length as parameter table and on the locations corresponding to the character parameters it should contain the code defining the parameter encoding. Possible encodings are:

STRENC_DFLT, STRENC_SB, STRENC_UTF8, STRENC_UTF16 and ***STRENC_UCS2***. Locations corresponding to non-character parameters should contain ***STRENC_DFLT*** or 0 (zero). For detailed description of the encodings please refer to the *MedSqlParEx()* function description.

Example

```
Indeks = 1000

&& Parameter 'John' will be passed and
&& binded as UTF8 string
MedSqlParA({indeks,20,MkStrUTF8('John')},
{0,0,STRENC_UTF8})

val = MedSelVal('SELECT sum(profit) FROM tab WHERE
emp=:3 and itemid=:1 and itemprice>:2')
```

See also: **MedSqlPar(), MedSqlParEx(), MedSqlPTrm()**

142. MedSqlParEx

Syntax:

MedSqlParEx(<*SQLpar1*>, <nSQLPar1Enc>, ...)

Extended version of the *MedSqlPar()* function. xHarbour/Harbour only. An extension allows to specify after each character parameter the encoding used for this parameter. This determines how the parameter value will be passed to the server. Currently, the following encodings are supported:

- ***STRENC_DLFT*** – previous parameter is encoded in default application codepage (set by HB_SETCODEPAGE()). Before sending to the server parameter is converted in the standard way (as in *MedSqlPar()*).
- ***STRENC_SB*** – previous parameter is encoded using single-byte encoding and will not be converted in any way
- ***STRENC_UTF8*** – previous parameter is encoded as UTF8 byte sequence and will not be converted in any way
- ***STRENC_UTF16*** – previous parameter is encoded as UTF16 byte sequence and will not be converted in any way
- ***STRENC_UCS2*** – previous parameter is encoded as UCS2 byte sequence and will not be converted in any way

The function allows to pass a character parameter for SQL statement in a chosen encoding and bind it in this encoding at a server side.

Not all encodings are possible for every database/configuration. For Oracle you can use any encoding. For MS SQL Server only STRENC_SB and STRENC_UCS2 are available.

Example

```
Indeks = 1000

&& Parameter 'John' will be passed and
&& binded as UTF8 string
MedSqlParEx(indeks,20,MkStrUTF8('John'),STRENC_UTF8)

val = MedSelValEx('SELECT sum(profit) FROM tab WHERE
emp=:3 and itemid=:1 and itemprice>:2')
```

See also: **MedSqlPar(), MedSqlParA(), MedSelStrVal()**

143. MedSqlPTrm

Syntax:

```
MedSqlPTrm( <lTrimSqlParams>.) -> lSuccess
```

By default, text parameters defined using MedSqlPar() and MedSqlParA() functions are transmitted to the server and used in SQL statement with eventual trailing spaces present at the end of parameter value. Using MedSqlPTrm(.T.) instructs Mediator server to trim all spaces present at the end of string parameters before using them in SQL statement. Calling MedSqlPTrm(.F.) restores the default behavior. Function returns .T. if succesfull.

See also: **MedSqlPar(), MedSqlParA()**

144. MedSrv64

Syntax:

```
MedSrv64() -> lMediatorServer64Bit
```

Function returns .T. if working with Mediator server compiled in 64-bit mode. Otherwise function returns .F.

145. MedSrvDate

Syntax:

```
MedSrvDate( <date> ) -> cRDBMSDateString
```

If the client is connected to the database server, the function returns string representing date in format, which can be used in the SQL expression. If there is no connection to the database, the function returns null string. The purpose of function is using it in SQL queries and filters.

Example:

```
USE qry AS ”SELECT count(*) FROM tab WHERE ndate =” +
MedSrvDate(UserInputDate)
```

146. MedSrvDay

Syntax:

```
MedSrvDay() -> nMediatorServerDay
```

One of the functions that read the date of logging into the server. It returns the day set on the server during the login.

147. MedSrvFltr

Syntax:

```
MedSrvFltr() -> lFilterOnServer
```

Function informs whether a filter set in the current workarea is evaluated on the database server. Function returns .T. when the filter was set by:

- SET SQL FILTER command or
- SET FILTER TO command and was correctly translated into SQL

Function returns .F. when:

- Filter was not set in the current workarea
- filter set by SET FILTER TO command is evaluated on client workstation.

148. MedSrvMnth

Syntax:

```
MedSrvMnth() -> nMediatorServerMonth
```

One of the functions that read the date of logging into the server. It returns the month set on the server during the login.

149. MedSrvSys

Syntax:

```
MedSrvSys() -> nServerOperatingSystem
```

The function returns the integer number that represents the system on which the Mediator server is operating:

SRV_SYS_NETWARE – one of Novell NetWare versions

SRV_SYS_NTX86 - Windows NT on PC platform (Intel)

150. MedSrvVMaj

Syntax:

```
MedSrvVMaj() -> nServerMajorVersion
```

This function returns the integer number that represents the main version number of the system on which the Mediator server is operating.

151. MedSrvVMin

Syntax:

```
MedSrvVMin() -> nServerMinorVersion
```

This function returns the integer number that represents the detailed version number of the system on which the Mediator server is operating.

152. MedSrvYear

Syntax:

```
MedSrvYear() -> nMediatorServerYear
```

One of the functions that read the date of logging into the server. It returns the year set on the server during the login.

153. MedStReset

Syntax:

```
MedStReset([<cUserComment>]) -> cDumpFileName
```

Writes the collected session statistics to the file and clears statistics counters. Returns the name of the server file where statistics data has been written or "" (empty string) if there was an error while writing statistics. After a call to *MedStReset()* the statistics are still collected. To turn off collecting statistics call *MedStStop().*

As an option you can specify *cUserComment* which will be written to generated statistics file. Written statistics can be analyzed using mkstat.exe command line utility.

See also: **MedStStart(), MedStStop()**

154. MedStStart

Syntax:

```
MedStStart()
```

Activates collecting by Mediator server of the session operations statistics. After statiscics are started, Mediator server stores the information about all the operations executed by this session. This includes the type of operations, number of times they are executed and total time of execution of each type of operation.

These statistics can be analyzed using mkstat.exe command line utility.

To collect statistics for all sessions connected to Mediator server use the appropriate interface present in *mmt.exe* tool.

See also: **MedStReset(), MedStStop()**

Example

```
MedStStart() && turn on collecting statistics
&& the profiled operations come here
fname = MedStStop("Invoice transaction statistics")
? "Statistics has been written to file " + fname
```

155. MedStStop

Syntax:

```
MedStStop([<cUserComment>]) -> cDumpFileName
```

Writes the collected session statistics to the file and turns off the collecting of session statistics. Returns the name of the server file where statistics data has been written or "" (empty string) if there was an error while writing statistics.

As an option you can specify *cUserComment* which will be written to generated statistics file. Written statistics can be analyzed using mkstat.exe command line utility.

See also: **MedStStart(), MedStReset()**

156. MedStTbsp

Syntax:

```
MedStTbsp( <cTablespaceName> )
```

For Oracle and PostgreSQL. The function specifies the name of a tablespace in which tables and indexes are to be created. The empty string resets to default tablespace for a particular user.

157. MedTabOwnr

Syntax:

```
MedTabOwnr( [ <nWorkareaNumber> ] ) -> cTableOwner
```

The function returns the owner (user on the database server) of the table opened in current or specified workarea.

158. MedTabTemp

Syntax:

```
MedTabTemp( <lTableTemporary> )
```

For Oracle only. After MedTabTemp(.T.) call all subsequent dbcreate() calls and USE .. AS 'SELECT..' SCROLLABLE calls will create tables with the clause GLOBAL TEMPORARY .. ON COMMIT PRESERVE ROWS. This means that all rows inserted to this table will be visible to the current session only, and will be removed on the end of the session. Memo fields are not allowed in such tables. All storage parameters specified by OraSt* functions are ignored.

159. MedTime

Syntax:

```
MedTime() -> nServerTime
```

The function returns the system time from the machine where Mediator server is running. The time is returned as number of seconds elapsed since midnight. Use the following expressions to obtain the hours, minutes and seconds of the current time respectively: INT(nServerTime/3600), INT(MOD(nServerTime,3600)/60) and MOD(nServerTime,60).

See also: **MedDate(), MedDateTm()**

160. MedTName

Syntax:

```
MedTName() -> cTableName
```

Function returns the name of the database opened in current workarea.

See also: **MedFTName()**

161. MedTrMode

Syntax:

```
MedTrRes(TR_MODE_MANUAL | TR_MODE_AUTO)-> nOldTrMode
```

The command sets the mode of functioning of drivers. Default mode is MANUAL (TR_MODE_MANUAL), in which all changes in database are committed automatically. In order to switch to transaction mode, a BEGIN TRANSACTION command is necessary to be executed every time.

The transaction mode is default in AUTO mode (TR_MODE_AUTO). In order to commit changes a COMMIT TRANSACTION command has to be executed every time. After each COMMIT TRANSACTION or ROLLBACK TRANSACTION the program will automatically switch to transaction mode without necessity of executing BEGIN TRANSACTION.

The function returns the old transaction mode.

The command SET TRANSACTION MODE is an equivalent of this function.

162. MedTrRes

Syntax:

```
MedTrRes()-> <nResult>
```

The function returns numerical value indicating the result of execution of BEGIN TRANSACTION, COMMIT TRANSACTION and ROLLBACK TRANSACTION commands. The function returns TRANS_SUCCESS if command was executed successfully or TRANS_ERROR otherwise. It is to be called immediately after execution of a command.

The function can also be called during transaction, after execution of other commands that cause changes in the database (for example UPDATE or DELETE). If errors are encountered, the transaction can be rolled back. After execution of operation a **MedTrRes** function can be called which returns one of the following values:

TRANS_SUCCESS – the operation was successful;

TRANS_ROLLBACK – the operation was unsuccessful and transaction was rolled back;

TRANS_TIMEOUT - the operation was unsuccessful due to unavailability of some resources and it was rolled back.

The list of commands which execution results can be checked by calling the **MedTrRes** function:

- SKIP
- GO TO

- FLOCK
- RLOCK
- UNLOCK
- APPEND
- COMMIT
- GO TOP
- GO BOTTOM
- SEEK
- USE

The more convenient alternative of acquiring status of execution of transaction is introducing SRV/1400 and NET/1301 error handling procedures (see errors description).

163. MedUMrkAll

Syntax:

```
MedUMrkAll( <nHandle> )
```

The function removes all records (markers) from the mark table identified by *nHandle*. Function execution is not buffered.

Example:

```
mhdl = MedMrkOpen( "mark01", "goods", "good_id")
IF mhdl == 0
   return
ENDIF
MedUMrkAll( mhdl )
MedMrkClose( mhld )
```

164. MySQLDbTyp

Syntax:

```
MySQLDbTyp( [<nDbTabType>] ) -> nPreviousTabType
```

Function sets the table type to be used for all subsequent MySQL table creation operations performed by Mediator. *nDbTabType* parameter specifies the desired type which can be one of the following (symbolic constants are defined in *Mediator.ch* header file):

MYSQL_DEFAULT_TT (0) - create tables using default MySQL Server mode (usually MyISAM type)
MYSQL_MYISAM_TT (1) – create tables as MyISAM type
MYSQL_INNODB_TT (2) – create tables as InnoDB type
MYSQL_BERKELEYDB_TT (3) - create tables as Berkeley DB type
MYSQL_HEAP_TT (4) – create tables as HEAP type (in-RAM tables)
MYSQL_ISAM_TT (5) – create tables as ISAM type
MYSQL_MERGE_TT (6) - create tables as MERGE type

Function returns previously active creation mode. Mode set using *MySQLDbTyp()* function is active until its next call. Called with no parameters, function returns the active table creation mode.

Example:

```
MySQLDbTyp(4)  && set creation mode to HEAP
dbreate(...)   && create table of type HEAP
MySQLDbTyp(0)  && return to default creation mode
```

See also: **MySQLMaxRc()**

165. MySQLMaxRc

Syntax:

MySQLMaxRc([<*nMaxHeapRecords*>]) -> *nPreviousMax*

Function defines the maximum number of records allowed for MySQL Server tables of type HEAP. This maximum value will be used for all subsequent HEAP tables created by Mediator in MySQL server. Parameter *nMaxHeapRecords* defines the maximum allowed numer of records for HEAP tables. Passing 0 (zero) value means there is no explicit limit and MySQL server default limit will be used (this is a default setting).

Function returns previously active limit or 0 (0 means there is no explicit limit). Limit set using *MySQLMaxRc()* function is active until its next call. Called with no parameters, function returns the active record limit or 0.

Example:

```
MySQLDbTyp(4)  && set creation mode to HEAP
```

```
MySQLMaxRc(100) && set record limit to 100
dbreate(...)   && create limited table of type HEAP
MySQLDbTyp(0)  && return to default creation mode
```

See also: **MySQLDbTyp()**

166. OraDefTbsp

Syntax:

```
OraDefTbsp( [<cUserName>] ) ->
cOracleDefaultTablespace
```

The function returns the default tablespace name for a specified Oracle user or an empty string. The empty string is returned if a current user does not have privileges to read the data from DBA_USERS view. To make the function useful, the current user ought to be given privileges to read from DBA_USERS view (a SELECT ON DBA_USERS privilege).

The function called without an argument will return the name of default tablespace for the current Oracle user.

167. OraIdxDBMS

Syntax:

```
OraIdxDBMS(<cTableName>,<lDBMSIndexes>) -> lResult
```

Oracle only functionality.

In normal operation, Mediator Server accepts from application and inserts expression index values for new and/or updated records. This function can be used to inform Mediator server that values for expression indexes defined for table *cTableName* will be supplied by the RDBMS pre-insert trigger manually defined by the user. Index expression values will be read back by Mediator server to ensure its proper operation.

The function can be called only for table which is not currently used (opened). The appropriate flag is stored in Mediator server repository, so when table is opened for the next time, the above setting is already active. *lResult* informs whether the flag has been successfully changed.

OraIdxDBMS(<cTableName>,.T.) – turns on RDBMS (trigger) index evaluation

OraIdxDBMS(<cTableName>,.F.) – turns on Mediator index evaluation

Example:

```
IF OraIdxDBMS("customer",.T.)
    ? "Customer table changed for manual (RDBMS) index evaluation"
ENDIF
```

See also: **OraRcnDBMS()**

168. OraRcnDBMS

Syntax:

OraRcnDBMS(<*cTableName*>,<lDBMSRecno>) -> lResult

Oracle only functionality.

In normal operation, Mediator Server calculates and inserts record numers (RECNO) for new records. This function can be used to inform Mediator server that record numbers for table *cTableName* will be supplied by the RDBMS pre-insert trigger manually defined by the user. New RECNO value will be read back by Mediator server to ensure its proper operation.

The function can be called only for table which is not currently used (opened). The appropriate flag is stored in Mediator server repository, so when table is opened for the next time, the above setting is already active. *lResult* informs whether the flag has been successfully changed.

OraRcnDBMS(<cTableName>,.T.) – turns on RDBMS (trigger) numbering

OraRcnDBMS(<cTableName>,.F.) – turns on Mediator numbering

Example:

```
IF OraRcnDBMS("customer",.T.)
    ? "Customer table changed for manual (RDBMS) record numbering"
ENDIF
```

See also: **OraIdxDBMS()**

169. OraStDeflt

Syntax:

OraStDeflt()

This function resets to defaults the storage parameters to be used for creating tables and indexes. Example:

See: **OraStPctFr**

170. OraStEIntl

Syntax:

OraStEIntl(*<nInitialExtent>*)

This function allows specifying the INITIAL parameter for tables and indexes created in the Oracle database. The INITIAL parameter designates size of an extent, specified in KB, allocated for an object during its creation.
Setting it to -1 means using default value for tablespace in which object is to be created.

Example:

```
USE cust
OraStEIntl( 500 )
OraStNext( 1024 )
OraStPctIc( 20 )
OraStEMax( 10 )
INDEX ON NAME TO KL_NAME
```

The INITIAL - 500 KB, NEXT - 1024 KB, PCTINCREASE - 20% and MAXEXTENTS - 10 parameters have been specified. Initial extent of 500 KB will be allocated for index KL_NAME that will be created. The next extent allocated after filling initial area will be of size 1024 KB. Each subsequent area will be greater by 20% then the previous one. Maximum number of extents allocated for that index will be equal to 10 (an attempt of allocation of the 11^{th} extent will result in an error).

171. OraStEMax

Syntax:

OraStEMax(*<nMaxExtents>*)

The function specifies MAXEXTENTS argument for indexes and tables created in Oracle database. The MAXEXTENTS argument specifies maximum number of extents allocated for an object. Specifying -1 effects in using default value for tablespace in which an object is to be created.

Example:

See: **OraStEIntl**

172. OraStEMin

Syntax:

OraStEMin(<*nMinExtents*>)

The function specifies MINEXTENTS argument for tables and indexes created in Oracle database. The MINEXTENTS argument specifies minimum number of extents allocated for the object during its creation.

Specifying -1 means using default value for tablespace in which object is to be created (usually 1).

173. OraStENext

Syntax:

OraStENext(<*nNextExtent*>)

The function specifies NEXT parameter for tables and indexes created in Oracle database. The NEXT parameter is size in KB of second extent allocated for an object.

Specifying -1 means using default value for tablespace in which object is to be created.

Example:

See: **OraStEIntl**

174. OraStPctFr

Syntax:

OraStPctFr(<*nPercentageFree*>)

The function specifies PCTFREE parameter for tables and indexes created in the Oracle database. PCTFREE is a percentage of free space that is to be left in the block allocated for storing given object (a table or an index) for later data modification. The free space is especially important for character columns that are filled little at the beginning but grow later.

The nPercentageFree parameter can contain values within range 0-99. The value equal to -1 resets PCFREE to default value (10).

Example:

```
RDDSetDefault( "MEDNTX" )
USE cust VIA ”DBFNTX”
OraStTbsp( ”DATA” )
OraStPctFr( 5 )
OraStPctUd( 70 )
OraStEIntl( 5*1024)
COPY STRUCTURE TO cust
OraStDeflt()
```

The structure of database CUST.DBF is copied to Oracle. Appropriate *cust* table will be stored in tablespace DATA. Initial extent of 5 MB is allocated for the table. The maximum degree of filling the block is 95 % (5 % of space is left for updating). The minimum degree of filling the block is 70 %. After table creation the storage parameters are reset to defaults.

175. OraStPctIc

Syntax:

```
OraStPctIc( <nPercentageIncrease> )
```

The function specifies PCTINCREASE parameter for tables and indexes created in Oracle database. PCTINCREASE is percentage by which each extent allocated after the second one (i.e. third extent and subsequent ones) is to be increased. The value equal to 0 effects in all allocated extents being the same as the second extent. Specifying parameter value equal to -1 effects in using default PCTINCREASE for tablespace in which table or index is to be created (default 50).

Example:

See: **OraStPctFr**

176. OraStPctUd

Syntax:

```
OraStPctUd( <nPercentageUsed> )
```

The function specifies PCTUSED parameter for tables created in Oracle database. PCTUSED means minimum percentage of space allocated in data block. The new records are inserted to given block if allocation percentage falls below PCTUSED. The nPctUsed parameter can be set within range 1-99.

The value equal to -1 resets the parameter to default of 40. The sum of PCTFREE and PCTUSED has to be lower than 100.

Example:

See: **OraStPctFr**

177. OraStTbsp

Syntax:

```
OraStTbsp( <cTablespaceName> )
```

The function specifies tablespace in which tables and indexes are to be created. The empty string resets to default tablespace for particular Oracle user.

Example:

See: **OraStPctFr**

178. ROLLBACK TRANSACTION

Syntax:

```
ROLLBACK TRANSACTION
```

The command rolls back the transaction previously began with BEGIN TRANSACTION command. The result of execution of ROLLBACK TRANSACTION command can be checked by calling **MedTrRes** function.

After rolling back the transaction all working areas in which changes were made are automatically refreshed by executing GOTO command to the record that was active during execution of ROLLBACK TRANSACTION command. In case of databases to which records were added with APPEND command the phantom record becomes active and eof() == .T.

WARNING!

If application enforces rolling transaction back by executing ROLLBACK TRANSACTION command, then the application itself is responsible for releasing all locks on databases and records. If the transaction is rolled back due to disconnection of the client and the server then all locks are removed automatically and the client is logged off the MEDIATOR server.

179. SET APPEND TIMEOUT

Syntax:

```
SET APPEND TIMEOUT [TO] [<nTimeout>]
```

By default, executing APPEND BLANK command can cause suspension of execution of the program if other workstation is executing the transaction that inserted the record into the same table. The APPEND BLANK command will wait until committing or rolling back the transaction executed by the other workstation. The SET APPEND TIMEOUT command changes default behavior. The *nTimeout* value specifies time in milliseconds after which the APPEND BLANK command will give up attempts of inserting the record into the locked table. In such case the NETERR() function returns .T. If APPEND BLANK command was executed within the transaction, **the transaction will not be rolled back**. The program may repeat attempts of inserting the record until achieving a desired effect without aborting transaction.

The SET APPEND TIMEOUT TO command without parameter resets behavior of APPEND BLANK command to default.

180. SET CLIENT CODE PAGE

Syntax:

```
SET CLIENT CODE PAGE [TO] MAZOWIA| CP852| CUSTOM
```

The command allows setting of the chosen code page on the client. This command complements SET SERVER CODE PAGE command. . You can create your own conversion between client and server code pages. To do this – please refer to CCONV.C example in the Mediator client distribution.

Conversions for Polish national characters (CP852 and MAZOWIA) are predefined.

181. SET FILTERING ON

Syntax:

```
SET FILTERING ON SERVER | CLIENT
```

Two modes of using filters were implemented: on the server (SERVER) or on the client (CLIENT). Usually filtering on the server is more efficient (fewer records are transmitted over network). Filtering on the server is set as default. If filtering on the server is chosen, the Clipper condition is translated onto corresponding SQL expression (translation only for Oracle; for other servers filtering is always done on the client – except of SQL filter). Sometimes it is impossible to translate the

condition. In such case filtering is located on the client workstation despite filtering being set on the server. The real mode of filtering can be checked by executing **MedFltRes** function after the SET FILTER ... command.

182. SET LOCK INTERVAL

Syntax:

```
SET LOCK INTERVAL [TO] <nInterval>
```

The command specifies the time in milliseconds after which the server process of serving the client will repeat attempt of locking a resource (a table with FLOCK or a record with RLOCK or DBRLOCK). The *nInterval* parameter should fit within the range of 20 to 30000 milliseconds. By default it is equal to 100 milliseconds. The parameter defined is valid for all later calls of FLOCK, RLOCK and DBRLOCK functions.

Example:

```
SET LOCK INTERVAL 1000
SET LOCK TRY 10
USE TABLE1 SHARED
time := seconds()
res := FLOCK()
?? ”After”
?? seconds() – time
?? ” TABLE ”
IF .NOT. res
    ?? ”was not ”
ENDIF
?? ”locked”
```

183. SET LOCK TRY

Syntax:

```
SET LOCK TRY [TO] <nTryCount>
```

The command specifies the number of attempts of locking the resource (table or record) when first attempt was unsuccessful. Attempts of getting the lock will be repeated *nTryCount* times until getting the lock or exceeding defined number of attempts. The *nTryCount* parameter should fit within the range of 1 to 10000 (default is 1). Defined parameter is valid for all later calls of FLOCK, RLOCK and DBRLOCK functions.

Example:

See: **SET LOCK INTERVAL**

184. SET PERFORATED NUMBERING

Syntax:

```
SET PERFORATED NUMBERING ON|OFF
```

The command sets the default mode of opening tables.

SET PERFORATED NUMBERING ON is an equivalent to the calling of MedSetPerf(.T.). It sets the mode of opening tables in a perforated (non-continuous) record numbering mode.

SET PERFORATED NUMBERING OFF is an equivalent to the calling of MedSetPerf(.F.). It sets the mode of opening tables in continuous record numbering mode (not allowing numbering "holes").

On an application startup, the continuous record numbering mode is active. This is a 100% CA-Clipper compatible mode.

See: **MedSetPerf(), MedPerfMod(), MedPerfRC()**

185. SET QUERY PRECISION

Syntax:

```
SET QUERY PRECISION [TO] [<nPrecision>]
```

The command specifies default precision for expressions read from the server in SQL query (USE AS ... command). If precision is not specified anywhere (i.e. neither in query nor with SET QUERY PRECISION command), results will have precision of 6 decimal points.

186. SET SERVER CODE PAGE

Syntax:

```
SET SERVER CODE PAGE [TO] CP852| MSWIN1250| CUSTOM
```

The command informs Mediator libraries what code page is used on the server. The command complements SET CLIENT CODE PAGE command. You can create your own conversion between client and server code pages. To do this – please refer to CCONV.C example in the Mediator client distribution.

Conversions for Polish national characters are predefined. For example to convert characters from Mazowia to MS Windows 1250 execute following commands at the beginning of the code of the program:

```
SET CLIENT CODE PAGE MAZOWIA
SET SERVER CODE PAGE MSWIN1250
```

To use your own conversion execute following commands at the beginning of the code of the program:

```
SET CLIENT CODE PAGE CUSTOM
SET SERVER CODE PAGE CUSTOM
```

187. SET SQL ERROR

Syntax:

```
SET SQL ERROR VERBOSE | SILENT
```

Setting SQL error handling in the VERBOSE mode results in every SQL error being displayed on the screen and aborting the program. The SILENT mode allows checking correct execution of SQL command with MedCmdRes() function.

Example:

```
SET SQL ERROR SILENT
MedExecSql("select a, b, c form tab")
IF MedCmdRes() != OTC_CMD_SUCCESS
   ? MedCmdRes(), MedErrText()
ENDIF
```

Above fragment of the program will display the code and message of RDBMS error since in SQL command a FORM word was used instead of FROM.

188. SET SQL FILTER

Syntax:

```
SET SQL FILTER TO <cSQLFilterCondition>
```

The command directly specifies SQL condition for WHERE clause.

Example:

```
SET SQL FILTER TO "check_code between 50 and 100"
```

A SQL filter is always set on the server. The knowledge of SQL syntax is required to use it. The syntax is described in manuals: *„ORACLE Server SQL Language Reference Manual"* – for Oracle, or *„Adaptive Server Anywhere Reference Manual"* – for SQL Anywhere.

Sometimes it can be convenient to use a name of a column that implements Clipper expression index. It is especially useful there, where expression index depends on the column and you want to use the column during calculation of condition because that can trigger significant performance improvement. In such case the name of an expression index can be preceded by IE$ prefix, for example:

```
SET SQL FILTER TO "IE$IW1 LIKE 'ABC%'"
```

Above command effects in getting only the records whose beginning of expression index from IW1 index is equal to "ABC".

189. SET TRANSACTION MODE

Syntax:

```
SET TRANSACTION MODE [TO] MANUAL | AUTO
```

The command sets the mode of functioning of drivers. Default mode is MANUAL, in which all changes in database are committed automatically. In order to switch to transaction mode, a BEGIN TRANSACTION command is necessary to be executed every time.

The transaction mode is default in AUTO mode. In order to commit changes a COMMIT TRANSACTION command has to be executed every time. After each COMMIT TRANSACTION or ROLLBACK TRANSACTION the program will automatically switch to transaction mode without necessity of executing BEGIN TRANSACTION.

The function MedTrMode() is an equivalent of this command.

190. USE <xQueryName> AS <cSQLSelect>

Syntax:

```
USE <xQueryName> AS <cSQLSelect>
[ALIAS <xqAlias>]
[NEW]
[SHARED>]
[EXCLUSIVE>]
[READONLY]
[PRECISION <nExpressionPrecision>]
[SCROLLABLE]
[PERMANENT]
[OVERWRITE]
[C1LOGICAL]
```

The command allows execution of SQL queries directly on the server and using results of query as if it was a table.

Example:

```
USE qry AS ”SELECT number,name,wage FROM employees”
NEW SCROLLABLE
BROWSE()
```

If keyword **SCROLLABLE** is used in **USE .. AS** command, the result of query is stored in a table named *xQueryName* that can be used like any other table. Otherwise result of the query is “virtual” table that can be browsed forward only with GO TOP and SKIP <positive_value> commands. It is disallowed to use other commands such as GO TO, SEEK or SKIP <negative_value > in such case.

If SCROLLABLE option is specified, the result of the query is placed in *xQueryName* table that is automatically opened. The table is deleted at the moment of closing it (USE). If the table containing result of query is to be stored, the PERMANENT option needs to be specified. After closing the query the table remains and it can be opened with following command:

```
USE xQueryName VIA "MEDNTX"
```

The PERMANENT option automatically implies SCROLLABLE option, so there is no need for explicit declaration.

Using OVERWRITE option results in automatic overwriting of existing table xQueryName. While the query is executed, the table cannot be used by other applications.

If browsing query backward is not necessary, avoid using SCROLLABLE and PERMANENT options since they result in additional server load (for Oracle it can require defining of very large rollback segment).

If C1LOGICAL option is specified, than all fixed length character fields (CHAR) with width 1 are treated as Clipper logical fields. The option is useful when a query is based on tables managed by Mediator and containing logical fields. If the option is not specified than these fields are visible as character fields.

SQL queries can significantly increase the speed of processing of time consuming calculations like grouping or summarizing.

The result of opening a *query* can be checked with **NETERR()** function. If the function returns .T., it is possible to check with **MedCmdRes** function whether the cause of failure was an RDBMS error (See: **MedCmdRes**)

Correct use of queries requires knowledge of SQL. Detailed description of SQL can be found in manuals: *„ORACLE SQL Language Reference Manual"* for Oracle or *„Adaptive Server Anywhere Reference Manual"* – for SQL Anywhere.

Query results could be referenced with name or alias from the expression list (after SELECT word). If the name on the list is not correct identifier, the default name composing of C and number of position on the list beginning from 1 is assigned to it.

Example:

```
USE sum AS "SELECT name, sum(wage) FROM wages GROUP BY
name"
? name, C2
```

or

```
USE sum AS "SELECT name, sum(wage) total FROM wages
GROUP BY name"
? name, total
```

Use an alias for table name and precede the name of an index with alias of table name, optionally with RDBMS username of owner of the table, the name of the table separated by dots and IE$ prefix to get values in SQL command from the column implementing expression index.

Example:

```
USE qry AS "SELECT x.name, x.surname FROM cust x WHERE
x.cust.IE$IW1 LIKE 'ABC%'"
```

Optional **PRECISION** clause specifies precision of results of query in columns containing numerical expressions. If only the name of the column is given as argument, the precision of result will be the same as precision of the column definition. The precision pertains only columns containing expressions.

Example:

```
USE sum AS ”SELECT sum(pay) paym FROM employees” NEW
PRECISION 2
```

In above query the **paym** column representing the sum of payments is returned with precision of up to two decimal points. In order not to specify precision in each query the default precision can be specified with following command:

SET QUERY PRECISION [TO] < *nExpressionPrecision* >

The default precision is valid for all queries executed after specifying it, unless the query contains PRECISION clause.

WARNING!

In case of executing queries on tables that correspond to Clipper databases, the SELECT command **is not** automatically completed with expression purging deleted records from the result set. If you need to use only records that are not deleted, add condition that will omit them:

WHERE is_deleted=’N’

Example:

```
USE query AS ”SELECT name, surname FROM employees
WHERE is_deleted=’N’”
```

Example:

```
USE query AS ”SELECT name, surname FROM emp WHERE
is_deleted=’N’”;
PERMANENT
USE
...
* opening result of query in another location in the
program
USE query VIA "MEDNTX"
```

For explanation how parameters can be specified for SQL statements, please see description of *MedSqlPar()* function.

191. USE <xQueryName> AS PROC <cProcName>

Syntax:

```
USE <xQueryName> AS PROC <cProcName> WITH <cParams>
[ALIAS <xqAlias>]
[NEW]
[SHARED>]
[EXCLUSIVE>]
[READONLY]
[PRECISION <nExpressionPrecision>]
[SCROLLABLE]
[PERMANENT]
[OVERWRITE]
[C1LOGICAL]
```

This command allows execution of PL/SQL stored procedure *cProcName* returning cursor. The procedure is called with *cParams* parameters specified in WITH clause. The first parameter of the stored procedure must be of ”REF CURSOR” type. When calling procedure this first obligatory parameter should be omitted from *cParams* parameters. As a result of command execution the workarea containing the cursor result is opened.

Other command options are identical to that described in USE AS SELECT command. For explanation how parameters can be specified for SQL statements, please see description of *MedSqlPar()* function.

Example:

```
Oracle (SQL*Plus):

create or replace package test_cur as
      type cur_type is REF CURSOR;
      procedure open_cur(cur in out cur_type);
end;
/
create or replace package body test_cur as
      procedure open_cur(cur in out cur_type) is
      begin
      OPEN cur for 'select username from all_users';
      end;
end;
/
```

```
xBase application:
   use qry as proc "test_cur.open_cur" with ""
   while .not.eof()
   skip
     ? username
   enddo
```

192. USE <xQueryName> AS FUN <cFunName>

Syntax:

```
   USE <xQueryName> AS FUN <cFunName> WITH <cParams>
   [ALIAS <xqAlias>]
   [NEW]
   [SHARED>]
   [EXCLUSIVE>]
   [READONLY]
   [PRECISION <nExpressionPrecision>]
   [SCROLLABLE]
   [PERMANENT]
   [OVERWRITE]
   [C1LOGICAL]
```

This command allows execution of PL/SQL stored function *cFunName* returning cursor. The function is called with *cParams* parameters specified in WITH clause. The function must return value of ”REF CURSOR” type. As a result of command execution the workarea containing the cursor result is opened.

Other command options are identical to that described in USE AS SELECT command. For explanation how parameters can be specified for SQL statements, please see description of *MedSqlPar()* function.

Example:

```
   Oracle (SQL*Plus):

create or replace package test_cur as
  type cur_type is REF CURSOR;
  function fopen_cur(stmt in varchar2) return cur_type;
end;
/
create or replace package body test_cur as
```

```
function fopen_cur(stmt in varchar2) return cur_type is
  cur cur_type;
  begin
    OPEN cur FOR stmt;
    return cur;
  end;
end;
/
```

```
xBase application:
   use qry as fun "test_cur.fopen_cur" with "'select
   username from all_users'"
   while .not.eof()
     ? username
   skip
   enddo
```

CHAPTER 8

VIII. Additional guidelines for designing applications

1. General recommendations

Opening tables with USE command

The preferred way of opening MEDNTX tables is to open them once, for example at the beginning of the program. That approach is different from the one typically used in XBASE/NOVELL applications where tables are opened only for data manipulation in order to minimize the risk of damaging them. In the RDBMS environment, it is best to open tables at the beginning of the program and keep them opened. This approach is fully secure and helps to save the time that is used for frequent opening and closing of tables.

Indexes

There can be only one simple index or one unique index (UNIQUE) on the column. Indexes with NEXT, WHILE, FOR and EVAL clauses are not implemented. Indexes in the SQL database are managed by the database server invisibly for the final user. The designer is only required to create or delete an index. All other activities are realized automatically by the server. Therefore **all** indexes are updated in case of changing the table, not only those specified in SET INDEX TO command. There is no need of reindexing data with REINDEX command.

2. Securing data from unauthorized access

Data stored in the SQL database is secured with standard protection mechanisms delivered by the server. Access to the particular user's account as well as to all data owned by that user requires the password. Access to data stored on another user's account is protected with different access privileges (see manuals: for example *„Oracle Server SQL Language Reference Manual"*).

The Mediator server makes it possible to store data of Clipper applications on any database user's account. User's identification can be performed by the database or Mediator server. In the first case, the user should enter the RDBMS username and the password. In the second case that identifies the user with the Mediator server as well as the database server, the user enters the username and the password for the Mediator server, and the server itself connects the user to RDBMS with information about the user and RDD passwords (see: The Mediator server) entered previously by the application administrator. For the purpose of porting an application, the first method is usually more convenient, i.e. identification with RDBMS. The second

approach is better for using an application, for data in the SQL database is better protected since the user has the access to data only through XBASE applications.

3. Working in the Wide Area Network (WAN)

The network traffic has to be minimized in order to make an application work well in WAN. The Mediator package delivers a built-in, very efficient coding of transferred data. This is why XBASE application using the Mediator can work in WAN at all. It is practically impossible with the typical architecture.

However, an application performance can be further enhanced by:

- Elimination of unnecessary data reading (SKIP)
- Using SQL queries for the execution of complicated calculations on the server
- Using a local file server for storing dictionary and temporary data in .DBF files
- Using compression available on routers
- Using TCP/IP protocol for client-server communication

And even better: use OTC Terminal software for effective work in WAN.

4. Limitations of the MEDIATOR

Indexes

The Mediator server fully implements expression indexes.

Due to a special feature of SQL servers, only one simple or simple unique index can be created on a particular column. Indexes with NEXT, WHILE, FOR and EVAL clause are not implemented.

SORT command

The SORT command is not implemented.

Appendix A

The list of reserved Oracle words. The following words must not be used as names of tables, columns or indexes.

ACCESS	EXCLUSIVE	NOAUDIT	SET
ADD	EXISTS	NOCOMPRESS	SHARE
ALL	FILE	NOT	SIZE
ALERT	FLOAT	NOWAIT	SMALLINT
AND	FOR	NULL	START
ANY	FROM	NUMBER	SUCCESSFUL
AS	GRANT	OF	SYNONYM
ASC	GROUP	OFFLINE	SYSDATE
AUDIT	HAVING	ON	TABLE
BETWEEN	IDENTIFIED	ONLINE	THEN
BY	IMMEDIATE	OPTION	TO
CHAR	IN	OR	TRIGGER
CHECK	INCREMENTAL	ORDER	UID
CLUSTER	INDEX	PCTFREE	UNION
COLUMN	INITIAL	PRIOR	UNIQUE
COMMENT	INSERT	PRIVILEGES	UPDATE
COMPRESS	INTEGER	PUBLIC	USER
CONNECT	INTERSECT	RAW	VALIDATE
CREATE	INTO	RENAME	VALUES
CURRENT	IS	RESOURCE	VARCHAR
DATE	LEVEL	REVOKE	VARCHAR2
DECIMAL	LIKE	ROW	VIEW
DEFAULT	LOCK	ROWID	WHENEVER
DELETE	LONG	ROWLABEL	WHERE
DESC	MAXEXTENTS	ROWNUM	WITH
DISTINCT	MINUS	ROWS	
DROP	MODE	SELECT	
ELSE	MODIFY	SESSION	

The list of Sybase Adaptive Server Anywhere 6.0 reserved words:

add	all	alter	and
any	as	asc	backup
begin	between	bigint	binary
bit	bottom	break	by
call	cascade	case	cast
char	char_convert	character	check
checkpoint	close	comment	commit
connect	constraint	continue	convert
create	cross	current	cursor
date	dbspace	deallocate	dec
decimal	declare	default	delete
desc	disable	distinct	do
double	drop	dynamic	else
elseif	enable	encrypted	end
endif	escape	exception	exec
execute	existing	exists	externlogin
fetch	first	float	for
foreign	forward	from	full
goto	grant	group	having
holdlock	identified	if	in
index	inner	inout	insensitive
insert	install	instead	int
integer	integrated	into	iq
is	isolation	join	key
left	like	lock	login
long	match	membership	message
mode	modify	natural	new
no	noholdlock	not	notify
null	numeric	of	off
on	open	option	options
or	order	others	out
outer	passthrough	precision	prepare
primary	print	privileges	proc
procedure	publication	raiserror	readtext
real	reference	references	release
remote	remove	rename	resource
restore	restrict	return	revoke
right	rollback	save	savepoint
schedule	scroll	select	session
set	setuser	share	smallint
some	sqlcode	sqlstate	start
stop	subtrans	subtransaction	synchronize

syntax_error	table	temporary	then
time	timestamp	tinyint	to
top	tran	trigger	truncate
tsequal	union	unique	unknown
unsigned	update	user	using
validate	values	varbinary	varchar
variable	varying	view	when
where	while	With	Work
writetext			

Appendix B

SQL scripts for the XBASE object manipulation in the Oracle database

In order to access data, which implement Clipper objects in an Oracle database, the Mediator package contains scripts for manipulating that data. Scripts can be executed with Oracle SQL*Plus. During installation of the MEDIATOR package, scripts are stored in an Oracle working directory so that they can be executed without specifying path to them. For example, to run a script that reads the information about tables (Clipper databases), use the following command:

```
SQL> @tabs
```

where tabs is the name of the script.

Scripts can be executed from the Oracle client workstation (for example, Windows 3.11, Windows 95 or Windows NT Server/Workstation). In such case, a set of scripts is copied to an appropriate directory in the client workstation (for example, ..\ORAWIN95\BIN in Windows 95).

The scripts delivered in a package

1. TABS.SQL

The script returns names of tables that implement Clipper (MEDNTX) databases available on the account of the current user. The number of columns and indexes of a table is returned.

Commandline:

```
SQL> @tabs
```

Result:

```
*****************************
* Clipper tables
*****************************
TABLE     Columns  Indexes
-------- --------- ---------
GT              1         0
AKWIZ          54         4
A_S             2         0
BANK           61         6
CARGO         222        11
```

```
PRICES           4         0
DECREES         56         5
DEBT            26        14

8 rows selected.
SQL>
```

2. TCOLS.SQL

The script returns column characteristics (i.e. database fields) of a table the name of which was given as an argument. The returned information includes the name of the column, the type (Clipper type), length and precision.

Commandline:

```
SQL> @tcols debt
```

Result:

```
*********************************
* Columns of table debt
*********************************
COLUMN      TYPE    LENGTH   PRECISION
---------- ------- --------- ---------
CODE       Numeric        10         0
NAME       Char           76
ADR        Char           35
...
RECIP      Char            1
EMPLOYEE   Char            1

26 rows selected.
SQL>
```

3. TIND.SQL

The script displays the characteristics of indexes created on the table which name was given as an argument. The information returned includes a Clipper index name, the name of the column on which the index is implemented, indexing expression and length of the column that implements the index.

Commandline:

```
SQL> @tind debt
```

Result:

```
*********************************
* Indexes of table debt
```

```
*********************************
INDEX    COLUMN   EXPRESSION                    TYPE          LENGTH
-------- -------- ------------------------- --------------------
DLU_KD   IE$0    D+STR(CODE,10)               Expression         11
DLU_ND   IE$1    D+NAZ                        Expression         77
DLU_MW   IE$8    W+STR(MA+100000000000,14)    Expression         15
DLU_SW   IE$9    W+STR(SALDO+100000000000,14) Expression         15

14 rows selected.
SQL>
```

The type of an index can be described as:

- `Simple`
- `Expression`
- `Simple unique`
- `Expression unique`
- `Simple descending`
- `Expression descending`
- `Expression fun.` – expression index; an expression contains the call to a user defined function
- `Expression unique fun.` – expression unique index; an expression contains the call to a user defined function
- `Expression descending fun.` – expression descending index; an expression contains the call to a user defined function

4. DROPTAB.SQL

The script deletes a table (Clipper database) which name is given as an argument from a current account of an Oracle user.

Commandline:

```
SQL> @droptab debt
```

Result:

```
************************************************************
* Warning! Deleting table debt!!!
* Continue? (Y/N) y
*
* Deleting table (y)
************************************************************
Deleting rows from table DEBT...
```

```
Deleting table DEBT...
SQL>
```

5. DROPALL.SQL

The script removes all the tables that implement Clipper (MEDNTX) databases from a current Oracle user's account.

Commandline:

```
SQL> @dropall
```

Result:

```
************************************************************
* Warning! Deleting all Clipper tables!!!
* Continue? (Y/N) y
*
* Deleting Clipper tables is in progress (y). Please wait!
 ************************************************************
Deleting rows from table GT...
Deleting table GT...
Deleting rows from table AKWIZ...
Deleting table AKWIZ...
Deleting rows from table A_S...
Deleting table A_S...
Deleting rows from table BANK...
Deleting table BANK...
Deleting rows from table CARGO...
Deleting table CARGO...
Deleting rows from table PRICES...
Deleting table PRICES...
Deleting rows from table DECREES...
Deleting table DECREES...
SQL>
```

APPENDIX C

Oracle privileges

The list of Oracle privileges that a user should have in order to be able to access tables stored on another user's accounts, depending on the kind of functions being called.

1. USE

In order to execute the command *USE another_user\TABLE1*, the user has to have the following privileges granted by *another_user* (table owner):

SELECT ON CLP_TABLES

SELECT ON CLP_TAB_COLS

SELECT ON *TABLE1*

SELECT ON *TABLE1*_MEMO (if MEMO fields are stored in the table)

The same privileges allow browsing the table *TABLE1*.

2. APPEND

In order to execute the command APPEND in the table *another_user\TABLE1*, the user has to have a privilege to open the table (see paragraph 1) and:

INSERT ON *TABLE1*

INSERT ON *TABLE1*_MEMO (if MEMO fields are stored in the table)

3. REPLACE, DELETE and RECALL

In order to modify (delete or restore) records stored in the table *another_user\TABLE1*, the user has to have a privilege to open the table (see paragraph 1) and:

UPDATE ON *TABLE1*

UPDATE ON *TABLE1*_MEMO (if MEMO fields are stored in the table)

4. PACK

In order to purge deleted records in the table *another_user\TABLE1*, the user has to have a privilege to open the table (see paragraph 1) and:

DELETE ON *TABLE1*

UPDATE ON *TABLE1*

UPDATE ON *TABLE1*_MEMO (if MEMO fields are stored in the table)

and a system privilege:

ALTER ANY TABLE (if MEMO fields are stored in the table)

5. ZAP

In order to delete all records from the table *another_user\TABLE1*, the user has to have a privilege to open the table (see paragraph 1) and the following system privileges:
DELETE ANY TABLE
DROP ANY TABLE
ALTER ANY TABLE (if MEMO fields are stored in the table)

6. DBCREATE

In order to create a table on another user's account, the user has to have the following privileges granted by *another_user*:
SELECT, INSERT ON CLP_TABLES
SELECT, INSERT ON CLP_TAB_COLS

as well as the following system privileges:
CREATE ANY TABLE
ALTER ANY TABLE
CREATE ANY INDEX
SELECT ANY TABLE

7. CREATE INDEX

In order to create an index to a table stored on another user's account, the user has to have the following privileges granted by *another_user*:
UPDATE ON CLP_TAB_COLS

as well as the following system privileges:
CREATE ANY INDEX
SELECT ANY TABLE
ALTER ANY TABLE
UPDATE ANY TABLE

8. DROP TABLE

In order to delete a table stored on another user's account, the user has to have the following privileges granted by *another_user*:
DELETE ON CLP_TABLES

as well as the following system privileges:
DROP ANY TABLE
LOCK ANY TABLE
DELETE ANY TABLE

9. DROP INDEX

In order to delete an index on a table stored on another user's account, the user has to have a privilege to open the table (see paragraph 1) and:
UPDATE ON CLP_TAB_COLS

as well as the following system privileges:
DROP ANY INDEX

10. MedMrkNew

In order to create and store a marker table for another user's table on his account, the user has to have a privilege to open the table (see paragraph 1), and the following system privileges:
CREATE ANY TABLE
ALTER ANY TABLE
CREATE ANY INDEX

11. MedMrkAdd, MedMrkDel

In order to add or delete a marker to or from a marker table of another user, the user has to have a privilege to open the table (see paragraph 1), and the following privileges:

INSERT, UPDATE, DELETE ON a marker_table

and the system privileges:
INSERT ANY TABLE
DELETE ANY TABLE
UPDATE ANY TABLE

12. MedMrkAll

In order to add markers to all records of another user's marker table, the user has to have a privilege to open the table (see paragraph 1), and the following system privileges:
DROP ANY TABLE

and one of privileges:
INSERT ANY TABLE or INSERT ON marker_table

13. MedUMrkAll

In order to delete all markers from another user's marker table, the user has to have a privilege to open the table (see paragraph 1), and the system privileges:
DROP ANY TABLE

14. MedMrkRemv

In order to remove another user ‘s marker table, the user has to have the following privilege:
SELECT ON CLP_TABLES
as well as the system privilege:
DROP ANY TABLE

Adaptive Server Anywhere permissions

The list of ASA permissions that a user should have in order to be able to access tables stored on another user’s accounts, depending on the kind of functions being called.

1. USE

In order to execute the command *USE another_user\TABLE1*, the user has to have the following permissions granted by *another_user* (table owner):

SELECT ON CLP_TABLES

SELECT ON CLP_TAB_COLS

SELECT ON *TABLE1*

SELECT ON *TABLE1*_MEMO (if MEMO fields are stored in the table)

The same permissions allow browsing the table *TABLE1*.

2. APPEND

In order to execute the command APPEND in the table *another_user\TABLE1*, the user has to have a permissions to open the table (see paragraph 1) and:

INSERT ON *TABLE1*

INSERT ON *TABLE1*_MEMO (if MEMO fields are stored in the table)

3. REPLACE, DELETE and RECALL

In order to modify (delete or restore) records stored in the table *another_user\TABLE1*, the user has to have permissions to open the table (see paragraph 1) and:

UPDATE ON *TABLE1*

UPDATE ON *TABLE1*_MEMO (if MEMO fields are stored in the table)

4. PACK

In order to purge deleted records in the table *another_user\TABLE1*, the user has to have a privilege to open the table (see paragraph 1) and:

DELETE ON *TABLE1*

UPDATE ON *TABLE1*
If MEMO fields are stored in the table user has to have DBA permission.

5. ZAP

In order to delete all records from the table *another_user\TABLE1*, the user has to have DBA permission.

6. DBCREATE

In order to create a table on another user's account, the user has to DBA permission.

7. CREATE INDEX

In order to create an index to a table stored on another user's account, the user has to have DBA permission.

8. DROP TABLE

In order to delete a table stored on another user's account, the user has to have DBA permission.

9. DROP INDEX

In order to delete an index on a table stored on another user's account, the user has to have DBA permission.

10. MedMrkNew

In order to create and store a marker table for another user's table on his account, the user has to have DBA permission.

11. MedMrkAdd, MedMrkDel

In order to add or delete a marker to or from a marker table of another user, the user has to have a privilege to open the table (see paragraph 1), and the following privileges:
INSERT, UPDATE, DELETE ON a marker_table

12. MedMrkAll

In order to add markers to all records of another user's marker table, the user has to have DBA permission.

13. MedUMrkAll

In order to delete all markers from another user's marker table, the user has to DBA permission.

14. MedMrkRemv

In order to remove another user ‘s marker table, the user has to have DBA permission.

APPENDIX D

Using PL/SQL procedures stored in the Oracle server from XBASE/Mediator program

A complete example of CA-Clipper/Mediator program, which calls procedures and functions from a package stored in the Oracle server is given below. The package contains the sample function that executes certain calculations. The result of computation are two values, one of which is numerical and the other character that will be stored in local variables of the package. The two remaining functions of the package provide access to the listed values. The method of using them is demonstrated by the following program:

```
#include "mediator.ch"
request medntx

* Executing PL/SQL procedure
MedExecSQL("begin
plsql_example.plsql_proc(1,'ABCDEFGHI'); end;")
IF MedCmdRes() != OTC_CMD_SUCCESS
   "Procedure execution error"
ELSE
*  Reading results of executing procedure
   use ww as 'select plsql_example.res1 r1 ,
plsql_example.res2 r2 from dual'
   ? r1, rtrim(r2)
   use
ENDIF
```

The MedExecSQL function calls **plsql_proc** procedure. The result of the procedure is read in the next step by executing the query calling **res1** and **res2** functions. The **res2** function returns VARCHAR2 type value which has maximum length of 2000 characters on the Oracle7 server. This is why when writing its result unnecessary spaces should be removed by calling **rtrim** function.

Contents of the package **plsql_example**:

```
create or replace package plsql_example as
   procedure plsql_proc(num number, chr varchar2);
   function res1 return number;
   PRAGMA RESTRICT_REFERENCES (res1, WNDS, RNDS);
```

```
   function res2 return varchar2;
   PRAGMA RESTRICT_REFERENCES (res2, WNDS, RNDS);
end plsql_example;
create or replace package body plsql_example as
   t1 number;
   t2 varchar2(5);

   procedure plsql_proc(num number, chr varchar2) is
   begin
     t1 := num+1;
     t2 := substr(chr, 1, 5);
   end;

   function res1 return number is
   begin
    return t1;
   end;

   function res2 return varchar2 is
   begin
     return t2;
   end;

end plsql_example;
```

Above example above demonstrates a method of accessing procedures that return parameters of OUT or IN/OUT types. Since OUT type parameters cannot be read directly, you should build a package that will contain the procedure with such a parameter and store a calculated variable in local variable of the package, and make it available through a function.

Similarly, standalone (not encapsulated in packages) functions and procedures can be called.

APPENDIX E

Porting applications using tables that have the same names in different directories

In order to port an application which uses databases (*.DBF) that have the same names in different directories, design an appropriate structure of RDBMS users.

For example, an application uses files BAZA1.DBF and BAZA2.DBF located in three directories: C:\APP\DIR1, C:\APP\DIR2 and C:\APP\DIR3. In order to port the application, create three RDBMS users: DIR1, DIR2 and DIR3. Move databases BAZA1.DBF, BAZA2.DBF from a directory C:\APP\DIR1 to an account DIR1. Move databases BAZA1.DBF, BAZA2.DBF from a directory C:\APP\DIR2 to an account DIR2 and databases from C:\APP\DIR3 directory to an account DIR3. In order to be able to use all databases, you have to grant appropriate database privileges to the user who will connect to a database from a XBASE/MEDIATOR application. A description of privileges and instruction how to grant them is located in Appendix C. Application can connect to an existing user's account of a database (DIR1, DIR2, DIR3), or create a new one. Privileges should be granted to a chosen user or a group of users.

Sample XBASE application does not require any modification. The following command references a table DB1 owned by user DIR1:

```
USE C:\APP\DIR1\DB1
```

The part related to the drive and directory structure without the last directory is omitted (C:\APP\). The last part of the directory structure is interpreted as a database username (DIR1 in this case).

An equivalent command can be as follows:

```
USE DIR1\DB1
```

In this way you can port only one level directory structure without changing a XBASE application. In order to port a multilevel structure, you have to flatten it first and create an appropriate set of RDBMS users.

APPENDIX F

Data migration and administration tools

This appendix describes tools included in the Mediator software. Those tools are designed for migrating data between *.DBF files and a database server (*Dbf2Med*, *Ntx2Med*, *Dat2Med*, *Dbf2Sql*, *Med2Dbf*), as well as for administration of Mediator objects created in a database server (*mdbu*). Programs are written in the XBASE language and the source code is located in SOURCE\UTIL and SOURCE\MDBU directories.

1. CDX2MED

The purpose of this program is moving *.CDX index files to the database server. Index files are to be transferred to RDBMS after creating empty table structures with *dbf2med* program and **before** loading data with *dat2med* program. Moving index files after loading data is possible, but it can be slow, especially if expression indexes are created. Indexes are created in RDBMS with MEDCDX driver.

Syntax

```
cdx2med dbfName [cdx1Name cdx2Name .. cdx16Name] [/CX]
```

Description

The argument is the name of single DBF file and names of CDX index files created on DBF file. All files should be located in current DOS directory and the empty structure of the DBF file created with *dbf2med* program has to exist in RDBMS. The *cdx2med* program reads keys of orders contained in order bags specified as arguments and creates appropriate indexes in the table dbfName in RDBMS. Moving orders defined in the CDX file with name the same as table *dbfName* is always attempted. Indexes are created on the table that is owned by the RDBMS user who was used by the *cdx2med* program during for connection during startup.

WARNING

If index keys of transferred indexes contain calls to user defined functions (UDF) or the program variables, it is necessary to add those functions and variables to the *cdx2med* program.

If “Undefined symbol” error is displayed during creating indexes followed by name of standard function from the clipper.lib library (e.g. DESCEND()), it is necessary to add appropriate REQUEST command to the *cdx2med* program (e.g. REQUEST DESCEND).

Options

/CX – This option specifies code page conversions to be executed by the Mediator software during transfer of text data (key indexes values) between client workstation and database server. If option is not specified, no conversion takes place. Appropriate character specified instead of X determines the type of conversion. At the moment, following conversions are available:

A - client MAZOWIA, server MSWIN1250

B - client MAZOWIA, server 852

C - client 852, server MSWIN1250

Example

```
cdx2med accounts.dbf /CA
```

Executing command results in creating orders in accounts.cdx on the account table in the database. Order keys will be read from file accounts.cdx from the current DOS directory. If index key values computed on the workstation are transferred from the client workstation to the database server, they will be subject to type A conversion (client MAZOWIA, server MSWIN1250).

2. DBF2MED

This program is designed for copying the structures of DBF tables to a database server. It allows to create **empty** tables in a database server that correspond to DBF files that you want to port. The tables are ported with an MEDNTX driver and can be used with it. Data stored in tables is ported with *dat2med,* preceded by porting structures with *dbf2med* program.

Syntax

```
dbf2med fileSpecification | directory [/O] [/L] [/CX]
```

Description

The argument is a pathname of a single DBF file that is to be exported to RDBMS, or the name of directory containing DBF files to export. All files in that directory, except *.DBT and *.NTX files, will be treated as tables and recreated in RDBMS if the directory name is specified in a command line. Files with extensions different from DBF will be ported as well. While porting tables with DBF extension, the extension itself is removed, and the table name in RDBMS does not contain any extension. In case of tables with an extension different than DBF, their extensions are concatenated to the name of RDBMS table after „_” (underline) character. For example:

accounts.dbf (DOS)	→	accounts (RDBMS)
quant.abc (DOS)	→	quant_abc (RDBMS)

All ported tables are created on a default RDBMS user's account, i.e. the user to whom *dbf2med* program connects during the start-up. The name of directory, preceding the name of a exported file, **is not** treated by *dbf2med* as the name of RDBMS user on whose account the table is to be stored.

Options

/O - if the option is not specified in a command line and RDBMS finds a table named like a table that is to be created, *dbf2med* program displays appropriate information and asks whether an existing table is to be overwritten. Specifying /O causes existing RDBMS tables to be replaced with a new empty structure without asking even if tables contain data.

/L – the option specifies the method of implementation of MEMO fields. Binary fields will be created if the option is not specified (for Oracle LONG RAW). Values stored in binary fields are not a subject to any conversion of a code page. Specifying /L leads to the implementation of MEMO fields as text fields (in Oracle LONG), the values of which are subject to a code page conversion executed by the Mediator software.

/CX – the option specifies which code page conversions should be executed by the Mediator software while sending text data between the client station and the database server. If an option is not specified, no conversion takes place. An appropriate letter given as X specifies the type of conversion. Right now the following conversions are allowed:

A - client MAZOWIA, server MSWIN1250

B - client MAZOWIA, server 852

C - client 852, server MSWIN1250

/FO – CA-VO – all the tables created in RDBMS will be of type OEM

/FA – CA-VO – all the tables created in RDBMS will be of type ANSI

Example:

```
dbf2med c:\dbs\accounts.dbf  /O
```

The command creates an empty *account* table on the default RDBMS user's account. If a table like that exists in RDBMS, it will be replaced with a new structure without any warning.

```
dbf2med c:\dbs
```

The command causes creation of empty tables in RDBMS, on a default user's account, corresponding to all files stored in c:\dbs directory, with the exception of *.DBT and *.NTX files. The program will ask whether it is to be replaced with a new structure, if the table exists in RDBMS.

3. NTX2MED

The program is designed for exporting index *.NTX files to a database server. Index files should be ported to RDBMS after creating empty table structures with *dbf2med* program, before loading data with program *dat2med*. Porting index files after loading data is possible, but it can be a slow operation, especially while creating expression indexes. Indexes are created in RDBMS with an MEDNTX driver

Syntax

```
ntx2med dbfName ntx1Name [ntx2Name ... ntx16Name]
[/CX]
```

Description

The argument is a single DBF file name and names of indexes (.NTX) created on DBF file. All files should be located in a current DOS directory and an empty structure of a specified DBF file should exist in database. *ntx2med* program reads a keys of indexes specified in a command line and creates appropriate indexes on the table *dbfName* in RDBMS. Indexes are created on the table that is owned by an user to whom the *ntx2med* program connected during the start-up.

WARNING!

It is necessary to link user defined functions (UDF) and variables to *ntx2med* program if calls to those functions exist in index keys.

If during creating indexes an „Undefined symbol" error message is displayed, after which the name of standard function from a clipper.lib library is displayed (for example, DESCEND()), it is necessary to add to *ntx2med* program appropriate REQUEST commands (for example, REQUEST DESCEND)

Options

/CX – the option specifies which code page conversions should be executed by Mediator software while transferring text data between the client workstation and the database server. If an option is not specified, no conversion takes place. An appropriate letter given instead of X specifies the type of conversion. At that moment,

the following conversions are available:

A - the client MAZOWIA, the server MSWIN1250

B - the client MAZOWIA, the server 852

C - the client 852, the server MSWIN1250

Example:

```
ntx2med accounts.dbf  acc_i1 acc_i2 acc_i3 /CA
```

Executing command results in creation of indexes acc_i1, acc_i2 and acc_i3 on the table *accounts* in a SQL database. Index keys are read from acc_i1.ntx, acc_i2.ntx and acc_i3.ntx files from the current DOS directory. If they are transferred between the client workstation and the database server, values will be subject to the type A conversion (client MAZOWIA, server MSWIN1250).

4. DAT2MED

The program is designed for exporting data from DBF files to the database server. Before exporting data, it is necessary to export table structures with *dbf2med* program. The data is transferred with an MEDNTX or MEDCDX driver that can be used for manipulating them later.

Syntax

```
dat2med fileSpecification | directory [/N] [/CX]
```

Description

The argument is a pathname of a single DBF file the contents (records) of which are to be exported to RDBMS or the name of directory storing DBF files, the contents of which are to be exported. In case of specifying the directory name, all files located in that directory with the exception of *.DBT and *.NTX files will be treated as tables, and their contents will be transferred to RDBMS. Contents of files with extensions different from DBF will be exported as well.

WARNING!

It is necessary to link user defined functions (UDF) and variables to *dat2med* program, if keys of indexes, existing on the table to which data is loaded, contain calls to those functions and variables.

If „Undefined symbol” error message is displayed during loading data and the name of a standard function from a clipper.lib library follows (for example, DESCEND()), it is necessary to add appropriate REQUEST commands (for example, REQUEST DESCEND) to *dat2med* program.

Options

/N - If an option is not specified in a command line, all records from *.DBF files are exported. Records are added to existing contents of a table in RDBMS. Specifying /N leads to porting only those records which have RECNO() numbers greater than a current number of records in a given RDBMS table at the beginning of the transfer.

/CX – the option specifies which code page conversions should be executed by Mediator software during transferring text data between the client workstation and the database server. If an option is not specified, no conversion takes place. An appropriate letter given instead of X specifies the type of conversion. At that moment, the following conversions are available:

A - the client MAZOWIA, the server MSWIN1250

B - the client MAZOWIA, the server 852

C - the client 852, the server MSWIN1250

Example:

```
dat2med c:\dbs\accounts.dbf  /CA
```

Executing the command transfers all records from c:\dbs\accounts.dbf table to RDBMS. Records will be added to optionally existing contents of an account table on the default RDBMS user's account. During character data porting, the code page conversion type A will be used.

```
dat2med c:\dbs /CA
```

Executing the command transfers records from all tables in c:\dbs directory to RDBMS. The records will be added to existing contents of appropriate tables on the default RDBMS user's account. During exporting character data, the code page conversion type A will be used.

5. MED2DBF

The program is designed for importing tables and data from RDBMS to DBF files. The program can get only those tables from RDBMS that were created with an MEDNTX or MEDCDX driver by an application or *dbf2med* program.

Syntax

```
med2dbf [tableName|*] [/I] [/O] [/CX]
```

Description

The argument is the name of the table that is to be transferred with contents from RDBMS or * (asterisk), which means that all tables stored on the current RDBMS

user's account are to be imported. Specified tables are transferred from RDBMS, and appropriate DBF files with contents are created in a current DOS directory.

Options

/I - /I option automatically creates appropriate indexes (.NTX) on a newly created DBF file based on index description from RDBMS.

/CX – the option specifies which code page conversions should be executed by Mediator software during transferring text data between the client workstation and the database server. If option is not specified, no conversion takes place. An appropriate letter given instead of X specifies the type of conversion. At that moment, the following conversions are available:

A - the client MAZOWIA, the server MSWIN1250

B - the client MAZOWIA, the server 852

C - the client 852, the server MSWIN1250

Example:

```
med2dbf * /I /CA
```

Executing the command causes importing all tables of a current user, together with their indexes from RDBMS. Appropriate DBF, DBT and NTX files are created in a current DOS directory. During character data porting, the code page conversion type A will be used.

6. MDBU

The version of DBU program working on data in RDBMS via MEDNTX driver. The program is designed for executing typical administration functions, such as creating new tables and indexes, modification, editing and browsing contents of existing tables in a database.

Syntax

```
mdbu [/S] [/CX]
```

Description

The functioning of the program is very similar to DBU program. Remember that opening a table in MDBU automatically opens all indexes set on it. It is necessary to link user defined functions (UDF) and variables to *mdbu* program if index keys of a modified table contain calls to them.

Options

/S – by default all tables opened by MDBU are opened in an exclusive mode. Specifying /S causes that tables are opened in the shared mode. Remember that creating an index on the database is possible only in the exclusive mode.

/CX – the option specifies which code page conversions should be applied by Mediator software during text data transfer between the client workstation and the database server. If an option is not specified, no conversion takes place. An appropriate letter given instead of X specifies the type of conversion. At that moment, the following conversions are available:

A - the client MAZOWIA, the server MSWIN1250

B - the client MAZOWIA, the server 852

C - the client 852, the server MSWIN1250

Example:

```
mdbu /CA
```

Executing the command starts *mdbu* program. During the work on tables, the code page conversion type A is applied.

APPENDIX G

Error codes generated by the Mediator libraries

The appendix describes error handling used with the Mediator drivers, and gives a detailed list of possible errors grouped according to functional subsystems.

In case of encountering an error, the Mediator drivers generate an error using conventions and techniques typical for XBASE applications. An object that contains description of an error is created, and a current error handling procedure is called (standard or defined by a user). Errors generated by Mediator drivers are uniquely identified by the name of subsystem in which an error was encountered (`Err:SubSystem`), and a detailed error code (`Err:SubCode`). 3 subsystems function in Mediator drivers:

- MED MEDNTX and MEDCDX drivers subsystem
- NET network communication subsystem
- SRV database server subsystem

During the creation of an object with an error description, the following fields are filled:

- `Err:SubSystem` the name of the subsystem
- `Err:SubCode` detailed error code
- `Err:Description` error description
- `Err:Operation` name of a function in which an error was encountered
- `Err:GenError` always zero (0)
- `Err:Severity` ES_ERROR (defined in Error.ch)
- `Err:CanDefault` .F.
- `Err:CanRetry` .F.

In case of using a standard error trapping procedure, the error message is displayed on the screen. The message is as follows:

```
Error SubSystem/SubCode : Error description [(additional info)]
: operation
```

After the word „Error” the name of a subsystem follows, detailed error code, error description, optional information in parentheses and the name of the function in which an error was encountered.

1. Errors reported by SRV subsystem

SRV subsystem errors are displayed if a problem associated with a database server (for example, an Oracle server) operation appears. A detailed code (SubCode) is given for every error.

`SRV/1400  Transaction rolled back`

When executing an operation in the transaction mode on a database, an error was encountered which caused the server to roll the transaction back. After the message, the description of an error that appeared on the server and caused rolling the transaction back is displayed. An application is able to use its own error trapping procedure and, therefore, it is able to fully control the transaction execution and optionally repeat it in case of an error. However, remember that rolling an active transaction back is also executed by the server in case of the network disconnection of the application with the Mediator server. A NET/1301 error is reported in such a case.

`SRV/1020  Server Error`

An error was encountered while executing operations on a database. A description of an error on the server is given in parentheses.

2. Errors reported by the NET subsystem

Errors of the NET subsystem are reported if a problem associated with network communication between an application and the Mediator server appears. A detailed error number is displayed for each error (SubCode).

`NET/1301  Network connection lost`

The network connection between an application and the Mediator server was broken. All work areas, in which tables are opened with MEDNTX or MEDCDX drivers, are closed automatically without saving any current changes. The transaction in progress on a database server is rolled back automatically, the Mediator server releases automatically all locks set by an application on tables and records. An application can connect to the Mediator server again with MedLogin()

function, however, after setting the connection up, all tables need to be opened again.

`NET/1302 Not connected to Mediator`

An application attempted to execute an operation that required the call to the Mediator server, but the connection was not set up earlier. You should connect to the Mediator server at the time of starting the application or by calling MedLogin() function.

`NET/1303 Unable to connect to Mediator`

During attempts of establishing a network connection to the Mediator server, a problem associated with the network communication or network environment initialization was encountered. More details are given in parentheses.

3. Errors reported by MED subsystem

Errors of MED subsystem are reported if some specific problem associated with the MEDNTX or MEDCDX driver appears. For each error, a detailed number is displayed (SubCode). Most of the messages contain additional information in parentheses as well as the name of a function in which an error appeared.

`MED/1006 Create error`

An error was encountered while creating an index. The index cannot be created.

`MED/1020 Invalid datatype`

The program used a value of incorrect type for a given action.

`MED/1023 Exclusive open required`

It was necessary to open the table in the exclusive mode. This mode is required, for example, by INDEX ON, PACK and ZAP commands.

`MED/1025 Invalid READONLY workarea operation`

The application tried to execute a command that cannot be executed on the workarea with READONLY attribute. For example, it is not possible to modify contents of tables opened in READONLY mode as well as result of SQL query (USE AS), unless SCROLLABLE or PERMANENT option is specified.

MED/1027 Number of supported orders exceeded

The maximum number of indexes allowed on the table was exceeded. The maximum number of indexes on one table is 20. Remember that all indexes created on the table, including ones not turned on explicitly with SET INDEX TO command, are always updated.

MED/1201 No active index

There was an attempt to execute a command that requires an active index but there was no such index. Activate an appropriate index using some command, for example, SET INDEX TO. An example of the command that requires an active index is the SEEK command.

MED/1251 Invalid parameters

The function was given incorrect parameters. The number, type or value of argument may be incorrect.

MED/1252 Out of memory

An attempt of allocating memory for the program was unsuccessful.

MED/1253 Internal limit exceeded

Internal limit was exceeded. The limit may be associated with the size of internal buffers of the Mediator client application, or maximum allowed values of function arguments.

MED/1254 General error

The meaning of an error is explained in an additional message included in parentheses.

MED/1255 Internal error

An error associated with internal Mediator structures inconsistency. This error should not appear during normal operation. However, it may appear as a consequence of another error.

MED/1256 Workarea not initialized

An attempt was made to execute the command that pertains work area, however, no area was opened or specified.

MED/1257 Invalid FORWARDONLY workarea operation

An attempt was made to execute a command that can not be executed on area with FORWARDONLY attribute. The example of

FORWARDONLY area is result of SQL query (USE AS) if a SCROLLABLE or PERMANENT option was not specified during opening.