

# MiniM Database Server Advanced Guide

Version 1.28

Eugene Karataev

<mailto:support@minimdb.com>

<http://www.minimdb.com>

April 3, 2017



# Contents

<b>1</b>	<b>Administration and Set-Up</b>	<b>7</b>
1.1	MiniM service description . . . . .	7
1.1.1	Windows version . . . . .	7
1.1.2	Linux version . . . . .	9
1.2	File minim.ini description . . . . .	10
1.2.1	Server Section . . . . .	10
1.2.2	Telnet Section . . . . .	13
1.2.3	Process Section . . . . .	13
1.2.4	Journal Section . . . . .	16
1.2.5	Mnemonic Section . . . . .	17
1.2.6	Login Section . . . . .	17
1.3	File minimdb.ini description . . . . .	19
1.4	File minimti.ini description . . . . .	21
1.5	Registry records made by installer . . . . .	22
1.6	Backup and Restore . . . . .	24
1.7	Handling lack of disk space . . . . .	30
1.8	Telnet echo . . . . .	31
1.9	MiniM Collation Editor . . . . .	34
1.10	MiniM license key usage . . . . .	38
<b>2</b>	<b>Devices</b>	<b>41</b>
2.1	Development with TCP device . . . . .	41
2.2	Development with CON device . . . . .	47
2.3	ATR mnemonic . . . . .	50
<b>3</b>	<b>Technical Articles</b>	<b>53</b>
3.1	minim.exe command line switches . . . . .	53
3.2	ZDLL module development . . . . .	56
3.3	ZDEVICE module development . . . . .	65
3.4	User-defined z-functions . . . . .	73
3.5	User-defined z-commands . . . . .	74

3.6	Processes accounts . . . . .	76
3.7	Routine Editor Keystrokes . . . . .	77
3.8	MiniM Server Connect . . . . .	79
3.9	MiniM Server Connect, ActiveX . . . . .	85
3.10	Import / export API . . . . .	94
3.10.1	Global import . . . . .	94
3.10.2	Block global import . . . . .	95
3.10.3	Routine import . . . . .	96
3.10.4	Bytecode import . . . . .	96
3.10.5	Globals export . . . . .	97
3.10.6	Block global export . . . . .	98
3.10.7	Routine export . . . . .	99
3.10.8	Bytecode export . . . . .	100
3.11	Routine Change API . . . . .	100
<b>4</b>	<b>CHUI Utilities</b>	<b>107</b>
4.1	%BACKUP . . . . .	107
4.2	%DBCLEAN . . . . .	110
4.3	%DBCRC . . . . .	111
4.4	%DBSIZE . . . . .	112
4.5	%GBI . . . . .	114
4.6	%GBO . . . . .	115
4.7	%GDIR . . . . .	116
4.8	%GI . . . . .	117
4.9	%GL . . . . .	118
4.10	%GO . . . . .	118
4.11	%GS . . . . .	120
4.12	%JOBTAB . . . . .	120
4.13	%JOURNAL . . . . .	121
4.14	%LOCKTAB . . . . .	122
4.15	%PERFMON . . . . .	123
4.16	%RCHANGE . . . . .	124
4.17	%RCOMPIL . . . . .	126
4.18	%RCOPY . . . . .	127
4.19	%RDELETE . . . . .	128
4.20	%RDIR . . . . .	129
4.21	%RESTART . . . . .	130
4.22	%RESTORE . . . . .	131
4.23	%RFIND . . . . .	133
4.24	%RFIRST . . . . .	134
4.25	%RI . . . . .	135

4.26	%RIMF	137
4.27	%RL	138
4.28	%RO	138
4.29	%ROMF	139
4.30	%RS	140
4.31	%RSAIN	141
4.32	%RSAOUT	142
4.33	%SHUTDOWN	144
<b>5</b>	<b>Macro Preprocessor</b>	<b>145</b>
5.1	Macro Routines	145
5.2	#define	146
5.3	Macro Comment	148
5.4	#else	149
5.5	#endif	149
5.6	#execute	150
5.7	#if	151
5.8	#ifdef	152
5.9	#ifndef	152
5.10	#include	153
5.11	#undef	154
5.12	Macro functions	155
<b>6</b>	<b>MiniMono</b>	<b>159</b>
6.1	MiniMono Architecture	159
6.2	Data structures	161
6.3	Direct calls	164
6.4	Calls back	165
6.5	Difference list	166
6.6	MiniMono CHUI Tools	167
6.7	MiniMono GUI Tools	169
6.8	MiniMonoX	170
6.8.1	MiniMono.VM properties	173
6.8.2	MiniMono.VM functions	176
6.8.3	MiniMono.VM events	188
6.8.4	MiniMono.ServerString properties	202
6.8.5	MiniMono.ServerString functions	202
<b>7</b>	<b>MiniMono for Android</b>	<b>205</b>
7.1	SDK Content	205
7.2	Building Application	207

7.3	Assets Synchronisation Utility . . . . .	210
7.4	Examples . . . . .	211

# Chapter 1

## Administration and Set-Up

### 1.1 MiniM service description

#### 1.1.1 Windows version

MiniM service is a central part of MiniM Database Server, control selected MiniM instance and all shared server objects. Service can be installed or reinstalled without MiniM instance uninstalling.

##### **Installation**

Run `mnmsvc.exe` with key `/install`. Key is used case insensitive. On installing can be displayed dialog window with message about successful installation. If this window does not required, need to add key `/silent`. This key is used case sensitive too.

Service installing using

```
mnmsvc.exe /install /silent
```

is recommended for automatic installing using special installers. On installing service application check key *InstallName* in section *Server* in configuration file `minim.ini`. For correct configuration of different MiniM instances need to check different instances have different names.

Installation examples:

```
w:\MiniM\bin\mnmsvc.exe /install  
w:\MiniM\bin\mnmsvc.exe /install /silent
```

### Uninstallation

Run `mnmsvc.exe` with key `/uninstall`. Key is used case insensitive. On deinstallation can be displayed dialog window with successful service deinstallation. If this window does not required, need to add key `/silent`. This key is used case sensitive too. If deinstallation check this service has not been installed, this key is ignored and is displayed dialog window with error message.

Uninstalling service with

```
mnmsvc.exe /uninstall /silent
```

is recommended for automatic uninstalling using special installers.

Uninstallation examples:

```
w:\MiniM\bin\mnmsvc.exe /uninstall
w:\MiniM\bin\mnmsvc.exe /uninstall /silent
```

### Service working

After installing service by default still working under special Windows account `LocalSystem` and runs manually. If need to run service automatically on computer start, it is required change settings in the Services applet in the Control Panel for service with name "MiniM Service for `INSTALLNAME`" where `INSTALLNAME` is a selected MiniM instance name. Change settings for this service to be run automatically.

MiniM allow to use several instances on the same computer and with different versions. For correct configuration all MiniM instances must have different telnet ports in the configuration files `minim.ini`.

### Start and stop service from command line.

To start service from command line use internal Windows command `net` with key `start`:

```
net start "MiniM Service for MINIM00",
```

where `MINIM00` is a MiniM instance name.

To stop service from command line use internal Windows command `net` with key `stop`:

```
net stop "MiniM Service for MINIM00"
```

This commands must start and stop appropriate services. If this installations have run any processes, this processes will terminates.

Examples how to start and stop MiniM service:



```
W:\MiniM\bin>net start "MiniM Service for MINIM00"
```

```
W:\MiniM\bin>net stop "MiniM Service for MINIM00"
```

Messages appeared on the screen are depended on Windows version and localization used.

To check MiniM service is run administrator can see Windows Service Manager or have to run MiniM Service Control utility `minimti.exe` which show server is run as a system tray icon.

### 1.1.2 Linux version

MiniM Installer and Uninstaller are made as a single executable file and one is use to install or uninstall (or upgrade available MiniM instance). This thow modea are differs by one parameter `-u`. Installer works as a pure CHUI application ffrom the command line and use only simple character input, so can be used from the local computer or can be run remotely.

MiniM Database Server architecture allow to install and run on one computer several MiniM instances and with different versions too. This MiniM instances work independently from each other.

#### Installation

Make installer file executable:

```
chmod +x setupminim...
```

Names of installer can differs from each other in depends of version and target processor.

Run installer

```
sudo ./setupminim...
```

#### Uninstallation

Make installer file executable and use the same version of installer as such as version of MiniM need to be uninstalled:

```
chmod +x setupminim...
```

Name of installer can differs from each other in depends of version and target processor.

Run uninstaller with the `-u` option:

```
sudo ./setupminim... -u
```

If current user does not have enough rights, installer outputs the message about this.

### **Start and stop server from command line.**

To start, stop or restart MiniM Database Server daemon need to be used appropriate shell scripts in the /bin directory of the MiniM installation:

```
start.sh  
stop.sh  
restart.sh
```

If MiniM will have any version-dependent changes, this changes must be located in this scripts.

If it is need, administrator must include start of the MiniM server into the OS start sequence in depends of OS version, using the script

```
start.sh
```

All internal logging MiniM server do in the /bin subdirectory, file minim.log.

## **1.2 File minim.ini description**

### **1.2.1 Server Section**

Key **InstallName** - MiniM instance name. Each MiniM Database Server installation on the one computer must have unique name. It is recommended to use names derived from MINIM and use English letters and digits. This name is used by MiniM service mnmsvc.exe and to create different shared objects to run several MiniM instances at the same time. Installation name must be up to 31 characters length. If it is specified more than 31 characters, only first 31 characters will be used. If instance name contains other character than English letter or digit, this character will be replaced to the "M" character. Example:

```
InstallName = MINIM00
```

Key **LogFileName** - name or full name of file for server logging. Value by default - minim.log and placed in the /bin subdirectory of MiniM installation. If file name specified is invalid file name, server use default value. Log file

can be deleted, copied without server and data fault. This file contains reglamented records about server start and stop and error messages. This records are not required for server normal work, but can be useful to solve problems occured. Key name is used case insensitive. Example:

```
LogFileFileName = w:/minim/bin/minim.log
```

Key **LogFileChunk** - log file chunk in bytes.

If minim.log file size need to be greater than log file chunk, MiniM rename file minim.log to minim.log.bak, minim.log file truncates to zero and next records add to the minim.log file. On this log file switchch procedure MiniM makes record about log file switching. Key name is used case insensitive.

If before log file switch file minim.log.bak exists, this file is removed. And in normal server working log file does not exceed two log file chunks by size. If MiniM administrator want to see log records by long time, this key value must can be increased. Example:

```
LogFileChunk = 100000
```

Key **BIJFileName** - name or full name of before image journaling file. This file is used by write daemon to write pages been changed before write to datafiles. This procedure prevents datafile from corrupting from hardware failure. This file is used by server start procedure to apply changes been made but not wrote to datafiles. Key name is used case insensitive. Example:

```
BIJFileName = w:/minim/bin/minim.bij
```

Key **ProcessLimit** - how much processes can work cuncurrently. If this value is not specified, server use value by default 2000. Value of this key does not affect to license limit and can be much more. Total process usage see in *License key usage* topic. MiniM administrator can manually decrease internal server allocations if use test license key with big capacity to test real application requirements. Example:

```
ProcessLimit = 1000
```

Key **Locale** - locale name is used by current MiniM instance. This is file nmae without extension of collation definition file name. This file is in the /nat subdirectory of MiniM installation and must have extension ".N". This collation definition is used by server for key comparisions, for upper and lower case characters. Example:

Locale = RUS

### MiniMono difference

MiniM Embedded Edition uses locale file name specified by the host process and file can be located in any directory.

Key **LockAreaSize** - count of memory in megabytes for locking table. This table is allocated once on server start. To change locking area server restart is required. Size of locking area can be determined by experiment. Minimal value is 1 Mb, maximum - 64 Mb. If application requires more than 64 Mb locking area it requires special MiniM build. This limits are intended to prevent unstable server state. If server requires big locking area, administrator must increase system swap and be sure that real physical memory is enough to prevent swapping caused by frequent operations. Example:

LockAreaSize = 8

### MiniM x64 difference

MiniM Database Server with x64 architecture does not use top limit for this value, administrator can set tens of gigabytes if need.

Key **AutoStartScript** name of file with MUMPS commands required execute on server start. If this key is specified, MiniM after full start executes commands from this file line-by-line. Commands are executed as entered manually by operator and before each line execution principal device makes a current device. This file is not routine, only lines of code. For example, if autostart.m file contains lines:

```
zn "user"
s ^TRACE($I(^TRACE))="start at " _$h
```

and if specified in key:

AutoStartScript = autostart.m

after start MiniM executes this lines as lines of MUMPS commands. This file cannot use dot syntax, but can call available compiled subroutines. File name cannot contain whitespace characters.

### 1.2.2 Telnet Section

Key **Start** specify need (value 1) or not (value 0) start internal telnet server on MiniM start procedure. If this key is absent or have any other value, internal telnet server does not starts. Example:

```
Start = 1
```

Key **Port** specify TCP/IP port number to use by internal telnet server. Can be used any value allowed for TCP/IP port. By default telnet servers and clients use port 23. MiniM administrator must check this port does not used by other software and different MiniM instances use different port numbers. If this key does not specified or is not valid TCP/IP port, MiniM does not start internal telnet server. Example:

```
Port = 2323
```

### 1.2.3 Process Section

Key **DeviceTableSize** specify the number of devices can be opened by one process in one time. This limit does not affect any license limit, it is only limit from uncontrolled devices opening by process on error and to prevent unneed memory usage. Increasing of this key does not have big memory wasting. Recommended value can be two of really used by one process. Minimal value is 4. If key contains not a number or less than 4, used 4 by default. Example:

```
DeviceTableSize = 8
```

Key **DeviceNameSize** specify maximum length of device name can be used. To select real length limit administrator must check device names can be used by process. Really big length can requires devices to run external processes with command line options. Ordinary, value of 1024 is enough. If process use device name with length exceeded this limit, this device will be created and properly function, but the *use* command can have unpredictable result to select device from devices available. It is not recommended make device length not enough. Exmample:

```
DeviceNameSize = 1024
```

Key **Storage** specify number of megabytes to use by process local variables. Minimal value is 1 Mb, maximum value is 64 Mb. If value exceed this limits, server select minimal or maximum values. This memory is allocated by each process and administrator must be sure real computer memory is enough. Example:

```
Storage = 8
```

Key **RoutineCache** specify number of megabytes for bytecode cache. Minimal value is 1 Mb, maximum is 64 Mb. If specified value exceeds this limits, server select one of appropriate limit. This memory is allocated once per server. Example:

```
RoutineCache = 8
```

Key **ReadLineRecallCount** specify how much lines of code can be stored in internal line editor for reuse by up and down arrows keys for TNT and CON devices. Minimal value is 10, maximum value is 128. This memory is allocated only once and only for specified devices. Example:

```
ReadLineRecallCount = 10
```

Key **ReadLineRecallBuffer** specify how much bytes need to be used to store for reuse by up and down arrows keys for TNT and CON devices. Minimal value is 80, maximum value is 2048. This memory is allocated only once and only for specified devices. Example:

```
ReadLineRecallBuffer = 1024
```

Key **FrameCount** specify number of stack frames limit per process. This memory is allocated once at process start and cannot be changed dynamically. For each function or subroutine call or for *recute* command process use one stack frame to save *new*-ed local variables and other stacked system variables. Minimal value is 16, maximum value is 131072. It is recommended to check real application frame count requirements and use plus several frames. Example:

```
FrameCount = 1024
```

Key **Namespace** specify database name by default to switch to on new process start. This database name is used by console, standard and telnet processes, and by internal service-run processes. If this database name does not specified, server select one of the following and in order: USER, %SYS. Databases are defined in the database configuration file minimdb.ini. Database name and key name are used case insensitive. Example:

```
Namespace = MAPSALES
```

Key **DBCACHESize** specify in bytes how much memory must be used for database pages cache on this server. This memory allocates once and is used by all database processes. Minimal value is 1 megabyte (1048576), maximum 1 gigabyte (1073741824). Example:

```
DBCACHESize = 104857600
```

Here server use for pages cache 100 megabytes. Note, that 1 kilobyte is 1024 bytes, 1 megabyte is 1048576 bytes and 1 gigabyte is 1073741824 bytes.

Pages cache is used for all databases of this server configuration.

#### **MiniM x64 difference**

MiniM Database Server with x64 architecture does not use top limit for this value, administrator can set tens of gigabytes if need.

Key **NullSubscripts** specify is null subscripts allowed on the server or not. If value is 0 - null subscripts does not allowed, and if 1 - null subscripts can be used. By default null subscripts does not allowed. If globals was created with null null subscripts, this data can fault on import in server where null subscripts does not allowed. By default MUMPS systems in generally does not allow null subscripts. Example:

```
NullSubscripts = 0
```

This setting is used for all existing and newly created local and global variables. Key name is used case insensitive.

Key **TrapOnEof** specify what must to do process if end of file occurred on reading from device, for example id device is a disk file. If this value is 0, process does not generate an error <ENDOFFILE> and sets system variable *\$eof* to value 1. Otherwise process generate an error <ENDOFFILE>. Value by default - 1. Example:

TrapOnEof = 1

Key **OnHalt** specify process action on terminate procedure related to current transaction context: rollback current transaction, commit current transaction or does nothing.

Table of supported values:

Commit	Commit transactions on each level
Rollback	Rollback entire transaction
other	Nothing to do with transaction context

Values are used case insensitive. If nothing been specified, process does nothing with current transaction context and all changes made are stored in database. This is behavior by default.

To choose value administrator must consider, in common case, if process will be terminates unexpectedly, server automatically run process guardian and guardian does not use this setting and roll back process transaction context and clears all locks created by failed process.

### 1.2.4 Journal Section

Key **JournalDir** specify subdirectory where server place journal files. Value by default is *journal*, this subdirectory is automatically created by installer. This disk must have enough space to store journal and administrator must check is disk space available to complete database operations. This key name is used case insensitive. Example:

JournalDir = w:/minim/journal

Key **JournalFileLimit** specify the limit of one journal file to grows before automatic switch to next journal file. Value is specified in megabytes. Minimal value is 1, maximum value is 1024. If value was not specified, server select maximum value, 1024 megabytes. Example:

JournalFileLimit = 200

Key **JournalCache** specify internal server cache size for journal records queue. Size is specified in megabytes. Value by default 8, minimal 1, maximum 64 megabytes. Key name is used case insensitive. This memory allocates by server only once and used by all processes. Example:



```
JournalCache = 8
```

### MiniM x64 difference

MiniM Database Server with x64 architecture does not use top limit for this value, administrator can set tens of gigabytes if need.

Key **TransactLevelLimit** specify maximum allowed transaction level. Minimal value is 1, maximum value is 32000. If value was not specified, sever use by default value 255. Example:

```
TransactLevelLimit = 255
```

## 1.2.5 Mnemonic Section

This section describes automatic mnemonic routines assignment for devices.

Key **CON** specify routine name to be an initial mnemonic routine for the CON devices of the server. If this value is not specified or is an empty string, server does not assign mnemonic routine for CON device automatically. This routine existence does not required to be assigned but is required in compiled form if any process will use mnemonics. Example:

```
CON = %CONX364
```

Key **TNT** specify routine name to be an initial mnemonic routine for the TNT (telnet) devices of the server. If this value is not specified or is an empty string, server does not assign mnemonic routine for TNT device automatically. This routine existence does not required to be assigned but is required in compiled form if any process will use mnemonics. Example:

```
TNT = %TNTX364
```

## 1.2.6 Login Section

Section *Login* specify need or not to run special mumps code on process run. This settings are applied to processes with potential user interaction: telnet processes, console or GUI connections. If nothing been specified, process starts work without execution any startup code. If was specified MUMPS commands, this commands executes before process goes to normal interactive state. To refuse user connection this code requires to execute the *halt* command. Before execution process goes to database specified in the section *Process*, Key **Namespace**.

Key **TNT** specify what commands must be executed first in telnet process.

Key **CON** specify what commands must be executed first in console process.

Examples:

```
[Login]
TNT = d login~%login()
CON = d login~%login()
```

This MUMPS commands can execute, for example, user login and password query, check user computernmae, change current database or to run assigned to this used command. If commands returns control without *halt* command execution. process stay in normal interactive state and wait user input after prompt.

Key **GUI** specify expression need to be evaluated to check user access rights. If this value is not specified or is an empty string, GUI applicaitons (MiniM Control Center, MiniM Global Editor and MiniM Routine Editor) does not query user login and password and continue to work.

If this value is not an empty string, this value is used as a MUMPS expression need to evaluate as an integer. If result is 0, GUI tools suppose connection failed, otherwise continue to work. Before execution code process switches to database specified in connection option and login and password are stored in special local variables with reserved names (%username and %password). After expression evaluation local variable %password removes. MiniM Database Server installer installs as example routine %logon which can be used to check user logins. It is not recommended to change routine %logon, because this routine can be changed by installer in next MiniM versions. Example how to use *GUI* key:

```
[Login]
GUI = $$VALID~%logon(%username,%password)
```

If this expression evaluates as 0, GUI utilities show message about login unsuccessful. If user not try other password, utilities exits. If expression evaluates as nonzero value, GUI utilities continue to work.

MiniM administrator can combine ini keys and routines to specify need login politics for users in compliance with application logins and check password check algorithms.

Values of the *TNT*, *CON* and *GUI* keys after installation are an empty strings and processes and GUI utilities does not ask user login and password. Installer on installing MiniM version over existing does not change this values. Values of this keys are read by depend of process logins and can be changed and affected without server restart.

### 1.3 File minimdb.ini description

Configuration file `minimdb.ini` store current database configuration. This is file of INI file format. Section names are database names case insensitive. For example, sections `[app]` and `[doc]` declare two databases APP and DOC. All databases must have valid names, first character must be English letter or percent (%) character with followed English letters or digits. Database name can be up to 31 characters in total.

MiniM Database Server implements internal builtin globals and routines mapping between databases. Routines with names starts with percent (%) and globals with names starts with percent phisically are stored in the %SYS database and are visible from any database. Globals with names starts with *mtemp* phisically are stored in the TEMP database and are visible from any database. This rules are mandatory and MiniM Database Server requires that databases with names %SYS and TEMP always be configured and mounted.

Each database section contains special key to specify database parameter. If MiniM detects unknown key in `minimdb.ini`, it generates appropriate diagnostic record in the `minim.log` file. All keys are used case insensitive.

Key **root** declare root datafile for database. Each database consist of one root datafile and optional sequence of extents. Minimal database structure is one root datafile. File name for datafile must be specified to be accessible to all MiniM processes which can work under different accounts. Files must be placed in accessible directories and this can be network or other virtual drives. Each datafile can have any size, MiniM Database Server uses only 64-bit addressing methods for positioning inside datafiles.

Key **SizeLimit** declares grows limit or the root datafile in megabytes. If MiniM reach this limit, database does not grows more or if present database extents, MiniM expand database extents specified. Example how to limit root file to 2000 megabytes:

```
SizeLimit = 2000
```

If this key (**SizeLimit**) does not specified, MiniM will extend this file without limits independently of was specified extents or not.

Key **AutoExpand** declare automatic database expansion (value 1) or expansion is forbidden (value 0). If database expansion is forbidden, MiniM Database Server does nor expand this database and any operation required to extend datafile will generate an error. Administrator can manually expand databases using utility `^%DBSIZE`. Database expansion is made automatically by special expand daemon process `minimed.exe`. Each datafile is expanded by 1 pages block. One page is 8 Kb and one pages block is 1 Mb, 128 pages.

Key **Mount** declare need to be used this database (value 1) at server start or not (value 0). Database can be configured but not in use by any reason. If database is not mounted (`Mount=0`), this database is not accessible.

Key **Readonly** declare database is accessible to read only (value 1) or can be changed (value 0). If database is readonly, any process attempts globals change will generate an error `<DBREADONLY>`.

Key **AutoCreate** declare need database be created on server (value 1) start or not (value 0). This key is automatically created by installer for the TEMP database. On server start all autocreated databases recreates from scratch, have size 1 Mb and have not any data.

Key **Journal** declare need (value 1) or not (value 0) this database be journaled. If this key is not specified, by default is used value 1. If journaling for database is disabled, all database changes made cannot be rolled back by rollback transaction and database state cannot be restored on restore from backup with journal records apply because journal does not contain any records fro this database. MiniM installer automatically desable journaling for the TEMP database.

Key **extentNNN** declare database extent as datafile for extent to be used with number **NNN**. Extent numbers must be numbered from 1 and follows by order (1, 2, 3, ...). Extend datafiles can be placed in any directory accessible by MiniM. For example:

```
extent1 = e:\minim\db\user1.ext
```

Key **SizeLimitNNN** declare grows limit for extent with number **NNN** in megabytes. Keys **SizeLimitNNN** must conform to **extentNNN** keys. If key **SizeLimitNNN** does not specified, this extent file will grows without limit.

If administrator configure database extents manually, first extent state must be file with zero bytes length. The state of extent is automatically controlled by MiniM Database Server and expanded by pages block 1 Mb size. Extents numbers must follows in order and last extent can have no limit to grows. If database consist of only root datafile, this root file can have no grows limit. Otherwise all files of database except last extent must have grows limit. MiniM Database Server use sequential expanding and filling up pages in database as logical pages sequence. Server does not expand and allocate new pages if database contains already freed pages.

Main extents appoinment is a different disks space usage by one database. On backup MiniM make backup and restore of pages as logical pages sequence. Backup uses datafile configuration at backup moment and restore uses datafile configuration at restore moment and this configurations can differs. So, extents are logical continuations of root datafile and prior extents.

In general, extent usage is not need if computer have enough disk space.

## 1.4 File minimti.ini description

File minimti.ini and minimti.exe application are used only in Windows editions of the MiniM Database Server. Unix-like MiniM versions use MiniM Launcher, windowed application to start and stop MiniM server and to run MiniM Client Tools.

Configuration file minimti.ini store settings for MiniM Service Controller minimti.ini. This application show an icon in system tray and popup menu to start, stop and restart MiniM Database Server and run additional applications. By default minim.ini is configured to run local GUI utilities, telnet and local MiniM console.

Popup menu have fixed part and configurable part. Configurable part of menu reads by minimti.exe application at start from special configuration file minimti.ini.

Application use only one section [Run]. All listed keys and values are menu items and command lines to start process. After installation section [Run] can be as follows:

```
Run Telnet Client = telnet localhost 23
Run MiniM Console = minim.exe
MiniM Routine Editor = minimre.exe
MiniM Global Editor = minimge.exe
MiniM Control Center = minimcc.exe
```

Here key **Run Telnet Client** is used for a menu item and *telnet localhost 23* as command line to start process.

Operator can delete or replace or add own items into minimti.ini. For example, add run minim.exe in console mode with automatic run specified commands:

```
My Application = minim.exe -x @mycmds.m
```

and specify in mycmds.m file commands to execute. This run mode execute commands and process still active until executes halts.

All changes made in minimti.ini are accepted only on MiniM Service Controller (minimti.exe) next start.

## 1.5 Registry records made by installer

Registry is used only in Windows editions of MiniM Database Server. Unix-like editions use ini files places in the directory

```
/home/username/.minim
```

and in subdirectories of MiniM installation.

Registry is used to store records necessary for service, installer and client utilities work.

### Records for service

One part of common MiniM Database Server architecture is a Windows service, which can be run, stopped, run automatically on computer start, and can be controlled locally or remotely. Records in registry made for a service are stored in a key:

```
HKEY_LOCAL_MACHINE\SYSTEM\  
CurrentControlSet\Services
```

Child records have name derived from "MiniM Service for " with followed MiniM instance name. For example, service for MiniM instance "MINIM1" use records

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\  
Services\MiniM Service for MINIM1
```

Registry records for MiniM service are made automatically on service installing and uninstalling. It is not recommended to change this records manually. This records are used by Windows to make operations with the service. Other registry records for a service are placed into subkeys:

```
HKEY_LOCAL_MACHINE\SYSTEM\
  ControlSet001\Enum\Root
HKEY_LOCAL_MACHINE\SYSTEM\
  ControlSet002\Enum\Root
HKEY_LOCAL_MACHINE\SYSTEM\
  ControlSet001\Services
HKEY_LOCAL_MACHINE\SYSTEM\
  ControlSet002\Services
HKEY_LOCAL_MACHINE\SYSTEM\
  CurrentControlSet\Enum\Root
```

and updated only automatically. Manual changes in this keys can break normal service functioning. If this occurs, MiniM reinstalling can repair registry records.

#### **Records for server installer**

MiniM Database Installer use special registry records as subkeys of:

```
HKEY_LOCAL_MACHINE\SOFTWARE\
  MiniM Group\MiniM\Instance
```

Child records have name derived from MiniM instance name, for example, MiniM instance with name "MINIM1" use records

```
HKEY_LOCAL_MACHINE\SOFTWARE\
  MiniM Group\MiniM\Instance\MINIM1
```

And here are stored records about installation directory, version, program group and other version-dependent records. This records are used by installer on installation or upgrade.

MiniM Database Installer also creates some registry records for uninstaller in the key

```
HKEY_LOCAL_MACHINE\SOFTWARE\
  Microsoft\Windows\CurrentVersion\Uninstall
```

with subkeys with name derived from MiniM, for example subkey MiniM\_is1.

### Records for client utilities

MiniM Client Utilities use registry to store connection settings, user settings, utilities appearance, for example colors and window position.

User settings are stored in as subkeys of

```
HKEY_CURRENT_USER\Software\  
  MiniM Group\MiniM\ControlCenter  
HKEY_CURRENT_USER\Software\  
  MiniM Group\MiniM\GlobalViewer  
HKEY_CURRENT_USER\Software\  
  MiniM Group\MiniM\RouEditor
```

and are differs for each user. This records are version-dependent, format and keys can be changed later. Removing this keys does not affect to client utilities work, in this case applications use default values for an each key and record.

Connection settings are stored as subkeys of

```
HKEY_CURRENT_USER\Software\  
  MiniM Group\MiniM\Connections
```

This key have child subkeys with name as connection name and values with connection options.

On removing all connections from registry client utilities will display an empty connection list and in this case operator must add at least one connection to connect to MiniM Database Server.

Records about connections to MiniM Database Server can be exported from the registry to a file and imported on the other computers to leave entering connection options on each client computer. This records have fixed format and can be changed by any application who can change records in the registry.

## 1.6 Backup and Restore

To make database system a fault-tolerance, MiniM Database Server implements backup and restore functionality and internal architecture things such as journal-based transactions and before image journaling.



MiniM Database Server administrator must understand MiniM details and principles how to plan database backup to make database system fault-tolerance.

MiniM Database Server contains several processes: service (mnmsvc.exe), write daemon (minimwd.exe), database expand daemon (minimed.exe), journaling daemon (minimjd.exe) and executive processes (minim.exe). Server starts and stops if starts and stops MiniM service. Service create and control common shared server objects, runs daemons and handles incoming telnet connections.

To protect common shared objects from unexpected termination of owners MiniM service guards all processes. Service wait all processes and if process terminates unexpectedly, service runs guardian process to clear internal structures, rollback uncompleted transactions or continue daemons.

On unexpected termination of executive process (minim.exe) service runs guardian process to clear internal shared objects (locking objects, modified cache pages). Guardian process does not use any license and terminates after clearing shared objects.

Second part of MiniM Database Architecture to make database system fault-tolerance is a datafiles set.

Main data of databases are in the datafiles, as configured in minimdb.ini file. Database is a contiguous file space made by root datafile and serial set of extents.

In addition to datafiles server use file of before image journaling minim.bij and journal files from journal subdirectory. Real files location defined in configuration file minim.ini.

All executive processes and daemons still work concurrwntly and asynchronously. Daemons wait changes made in internal queues to write to disks and if write condition is occured, daemons executes writing. Executive process does not make any writes to datafiles or to a journal and make read only of data from datafile into page cache. To implement concurrent access MiniM Database Server use internal strategy "one write, many read".

If MiniM Database Server stop working, after last executive process terminates, some part of data can be placed not in the datafiles, but in the before image journaling. On next server start MiniM Database Server executes special action to continue write data to appropriate datafiles to make global's data read correct.

Caution: MiniM Database Server implements logically correct data for executive processes only (minim.exe), not for data files.

MiniM start procedure provide special guaranteed control point of full database integrity, for which state the before image journal does not contain any data not written to datafiles. Server stopping procedure contains this control point too. But integrity control point is not guaranteed from power or hardware fault. To protect data from this fault MiniM Database Server support backup.

In general, all backup methods can be divided into two big groups - cold and hot backup. Cold backup is simple file backup with datafiles set copying and backing up for later restore as is into the same places. Hot backup is a special procedure for special files production without server stopping and to use later this produced files in special restore procedure. Cold backup can be divided to two groups - with server stopping and without server stopping. For MiniM Database Server cold backup without server stopping is not guaranteed method to get all need data into datafiles set.

To use cold backup with server stopping for MiniM Database Server it is recommended two methods: 1) stop server and copy datafiles set and before image journal or 2) stop server, start and stop again to prevent all unwritten before image journaling data writes to a datafiles. In first case before image journal is necessary to implement control integrity point. In second case control integrity point is guaranteed by server start procedure and datafiles contains all saved data. Journal files can be copied in depending of selected database restore strategy.

Journal is one more thing to implement some database functions. Journal in general include all files are placed into the journaling subdirectory. This directory location can be redefined by administrator in the configuration file. All files in journal subdirectory are used in total only, all files set.

MiniM Database Server writes into journal all need information about database changes. This records are used for two tasks: 1) transaction rollback and 2) database restore.

Files of journal on globals changes are continuously grows, unlike of database files. If processes changes only several global names, common datafiles size can no grows, but each of this action adds some record into journaling files. Database journaling can be disabled in database configuration file `minimdb.ini` for each used database personally.

Unlike of database files, who store current globals state, jornal stores all changes made like a time machine. This capability can be used by database administrator to create a database backup and restore strategy. Later is described the hot backup only and a hot backup details. After full backup been

made all databases continues to changes and all changes made are recorded into a journal. And, on database restore from backup file databases will be restored to backup moment. All changes made later can be applied from journal on database restoring. In other word, backup files and journal files are full files set to restore database. Administrator must understand that journal loss cannot restore database to last state but only to backup moment.

MiniM Database Server implements two hot backup forms - full and differential.

Full backup have three stages: 1) save to backup file all pages of datafiles and mark pages as backed up, 2) save to backup file all pages been modified (in time of first stage) and mark pages as backed up and 3) freeze all other processes, save all modified pages (in time of second stage) to backup file and mark as backed up and unfreeze freed processes. After backup server still work in normal execution.

On backing up datafiles pages MiniM uses some compression to reduce used space and backup file can be less then datafiles.

Differential backup make only second and third stages and save to backup file only pages been modified since prior backup - full or other differential. On restore it is required select backup files in the same sequence files been created - full and next set of differential backups. After restoring from last backup file operator can apply changes made and recorded in a journal to get last available actual database state. This procedure require, of course, journal files are correct. In this case MiniM search in a journal appropriate backup point and apply changes made from journal in the same order this changes been made including transactions rollbacks.

So, on datafiles restore operator must select one of possible alternatives about journal: 1) apply journal records after backup been made, 2) nothing to do with journal and 3) rollback transactions still active on backup moment.

Second alternative can be recommended if it is restoring from the backup files set and it is not last backup file.

Third alternative, with rolling back uncommitted transactions, creates logically correct datafiles on backup moment only. This alternative can be used if data changes been made after backup are unneed. This operation require journal records, MiniM must find backup point in a journal and rollback changes been made.

To reduce used by journal space operator can select on backup option to truncate journal. In this case MiniM removes from journal files set all records unneed to restore current uncommitted transactions. Some of journal files

MiniM can delete or replace. Journal truncation may take some time and MiniM have special protection for journal truncation - if hardware fails on truncation, at the next start MiniM complete truncation automatically.

Also operator can truncate journal files set at any time without backup procedure. This can be a part of selected fault-tolerance strategy.

Journal is a special linked sequence of journal files and MiniM uses in most cases only last of ones. Operator can execute special operation to switch current journal file. In this case MiniM server creates one more journal file and prior other journal files can be copied to another place or can be saved with a backup files. Journal files have special names as date file been created and next number for file created within this day. Operator can use journal files names to copy or move currently unneed journal files. It is recommended that operator be sure prior journal files really does not need and does not contains records need to rollback uncommitted transactions.

On restore MiniM check backup file type and restore type.

On restore MiniM check than server does not have any processes in a opened transaction state. In this case MiniM refuse from restoring to protect databases, transactions and to prevent transactions overwrites data restored from backup.

Backup files contains database size been at the backup moment. It is common datafiles size of database, including root datafile and all extents. On restore MiniM changes database size to saved in the backup file.

MiniM restore database size and change datafiles pages using currently used at restore moment configuration, specified in minimdb.ini file. Operator can backup database in one datafiles configuration and restore in another datafiles set. If need, MiniM can extend datafiles or shrink to zero size using current configuration. And on restoring all datafiles pages been backed up used as a logical sequence covered all database from root to last extext.

MiniM Database Server allow operator select special database restoring strategy, application-oriented.

Let it be two (of more) possible backup and restore strategies.

First strategy - "all changes been made are important and need to be transferred to other server". Operator must prepare common globals, routines and bytecode in a database and must make full backup of selected database with journal truncation. This backup file need to restore on other server with option dont use journal records. After restoring on target server we have database made on first server with all need routines, globals and

compiled bytecode. Also data can be transferred using export and import procedures, but in this case database will contain other data exists prior import data to transfer. And using backup-restore procedure we got the same database but in possible other datafiles configuration.

Second strategy - "all changed are important on last fault moment". Operator select backup periodicity of full backup, for example, once in a week. Full backup executes with journal truncation. And using small interval (for example, one day) operator plans differential backups without journal truncation. And, after full period (in this case one week) operator have one full backup and a set of differential backup files. And on restore operator can restore from full backup and set of differential backup files. All backup files except last must be restored without any journal operation and last differential backup file must be restored with applying journal records been made after last differential backup.

MiniM Database Server does not record in backup files any info about prior backups been made. And backup files does not contains any information about each other. So MiniM cannot control right order to restore database from backup files. This order must check operator. And, operator must create rules for backup file naming and restore strategy. MiniM have no any limitation about file names and extensions.

Each backup operation creates one more record to a journal and on restoring MiniM searches this backup point.

Operator can backup and restore databases using `^%BACKUP` and `^%RESTORE` utility or create own automated utility using MiniM internal functions `$view("db")`. And, this backup and restore operations can be embedded into application and implement special application requirements and strategy.

On hardware or software fault operator can combine three actions - solve problems with hardware or software, make backup and make restore database. Let it be, for example, some situations.

1) Corrupted one or more extents of database or before image journal and journal is correct.

Solution - restore database from backup with applying journal records.

2) Journal corrupted.

Solution - remove journal and make full backup.

3) Datafiles and journal are corrupted.

Solution - restore database from backup without journal applying.

4) Hardware fails on database restoring.

Solution - solve hardware problem and repeat database restoring.

5) Before image journal (minim.bij) file is corrupted and file minim.log contains a record about server (MiniM instance) startup problem.

Solution - stop server (MiniM instance), remove before image journal file (minim.bij) and start server (MiniM instance) again.

After restoring database from backup it is recommended check the nature of data been lost from the point of view of available applications. In any case it is recommended to use uninterruptable power supply and periodically check filesystem.

## 1.7 Handling lack of disk space

If MiniM Database Server detect that the disk space is not enough to complete operations, server launch waiting space been freed. Server attempt to write to minim.log file special diagnostic message and process who must write data generates an error about database operations cannot be complete. MiniM server detect the following disk space lack:

### **Configured database growth limit.**

If server process detects database size limit is reached, but some operation requires to extend database, process generates an error <DBLIMIT>.

It is recommended to analyze application been executed and make changes in application or in database configuration to conform application requirements.

### **Disk space is not enough to write to before image journal file minim.bij.**

If MiniM write daemon detect that disk space is not enough to complete write to before image journal operation, write daemon make special diagnostic record to minim.log file and switches all used databases into read only state. After this all MiniM processes on global change operations generates an error <DBREADONLY>.

If this situation has been detected, server have several pages of datafiles in internal page cache area. Write daemon periodically attempt to write pages to before image journal to complete database changes made. It is recommended to free some disk space for minim.bij file growth and using functions  $\$v("db",23)$  or  $\$v("db",24)$  switch databases from read only state

to normal, as specified in configuration file `minimdb.ini`. This changes can be made without server restart and `minimdb.ini` files changes are unneeded. After moment writing to before image journal can be possible write daemon completes database changes been made and continues to work. All other server processes can continue to work and does not lost transactional context or data.

#### **Disk space is not enough to expand database.**

If MiniM detect that database extending is required but disk have not enough space, expand daemon send signal to processes who waits database extending and this processes generates an error `<DBEXTEND>`. After this expand daemon switches databases into read only state.

It is recommended to analyze application requirements and disk space available and free disk space up to several megabytes (databases are expanded by 1 megabyte per step). After freeing disk space it is required using functions `$v("db",23)` or `$v("db",24)` switch databases states from read only to a normal state. After this changes all next database operations will be executed normal and processes can continue to work without transactional context and data been lost.

#### **Disk space is not enough for journal.**

If MiniM server detect that journal operation cannot be complete on disk space limit, journal daemon switches databases into read only state, writes special diagnostic message to a `minim.log` file and periodically attempt to complete journal operations. In this case internal journal buffer can contain journal records have not written to journal file. If administrator stops server, this journal records can be lost.

It is recommended to free disk space for journal subdirectory, for example, move old journal files into other place, and using functions `$v("db",23)` or `$v("db",24)` switch databases to a normal state.

In general, can be recommended configure databases growth limit as really need, periodically make backup with journal truncation and monitor disk space is available for MiniM server.

## **1.8 Telnet echo**

If you use default Windows telnet client (`telnet.exe`) to connect to MiniM Database Server, it can appear with local symbols echo on key press. It is a client settings, not a server. To disable symbols echo, you must connect to

any telnet server, for example MiniM, and press in the Windows telnet client window keystroke

```
Ctrl+]
```

This comand switches telnet client to local settings control. Next type this comand:

```
unset LOCAL_ECHO
```

End press Enter key twice. Telnet client will return to normal screen to interact with telnet server and store settings for later usage.

Full Windows telnet commands is listed by comand:

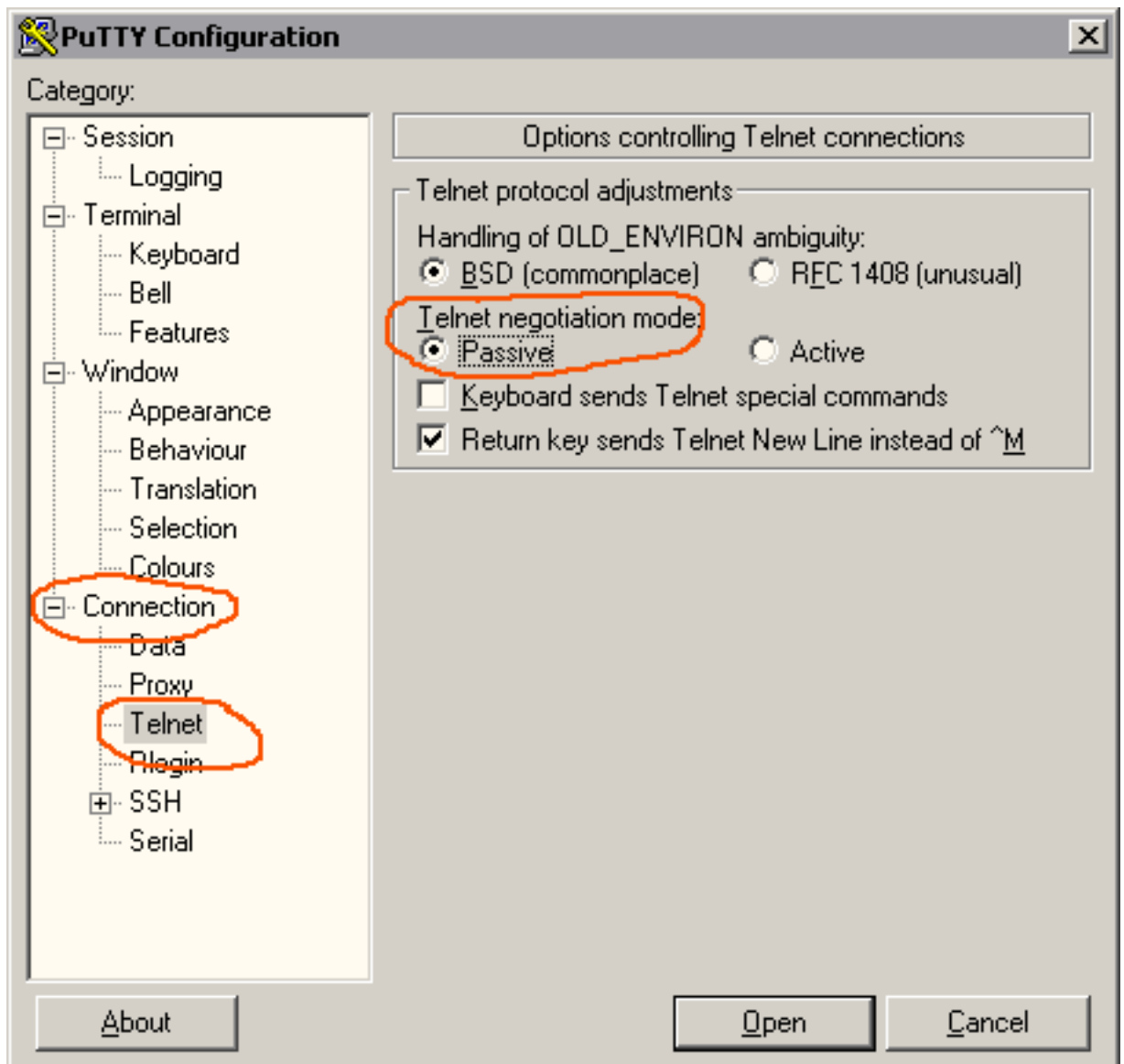
```
?
```

You can enable or disable any option available by the *set* and *unset* comands. Full comand options supported is listed by comand:

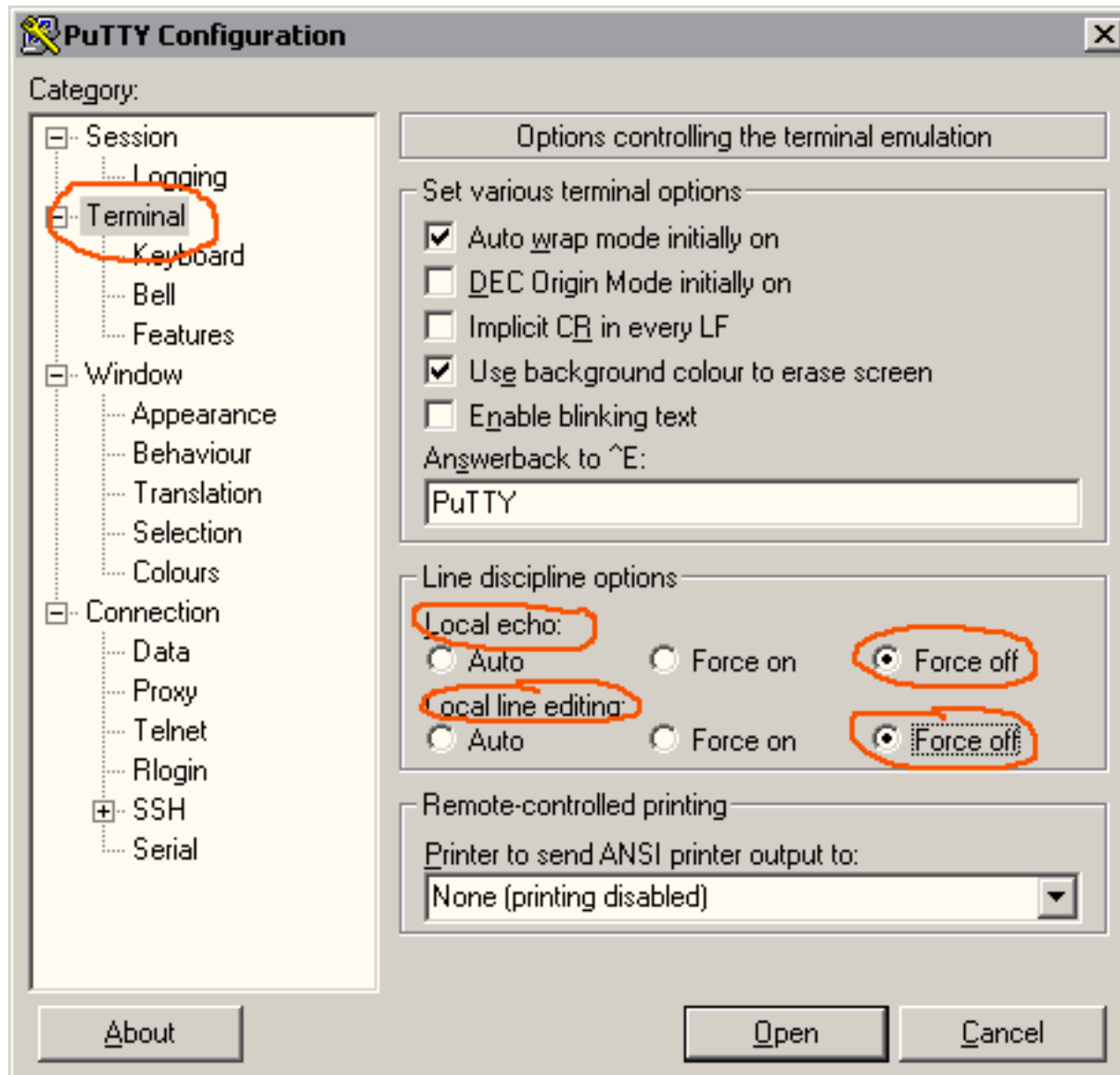
```
set ?  
unset ?
```

If you use Putty telnet client, you can see characters doubling too. To disable characters doubling as local echo, go to the Putty settings and in configuration tree select *Connection*, next select *Telnet* and for telnet options select *Passive*. This screen show how it appears:





Also you must select one more option for echoing. In configuration tree select *Terminal* and in both options *Local echo* and *Local line editing* select option *Force off*. This screen show how it appears:



If you are using other telnet client and this application make local echo with characters doubling, consult with support of this application how to configure this telnet client to disable local echoing.

## 1.9 MiniM Collation Editor

MiniM Database Server contains special GUI utility MiniM Collation Editor (/bin/minimne.exe) to edit or create new special collation definition files.

Collation definition contains in the file with .N extension and this file

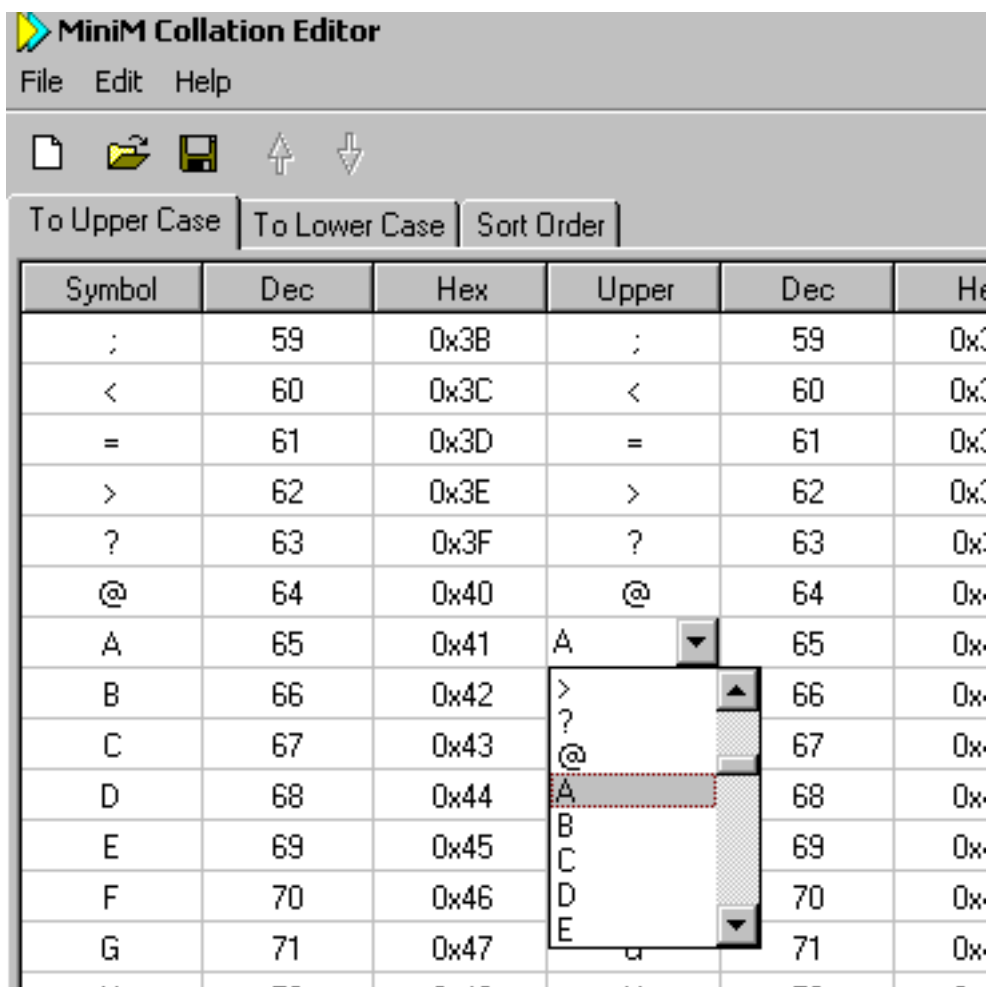
must be placed in the /nat subdirectory of MiniM installation. This file contains three definition tables:

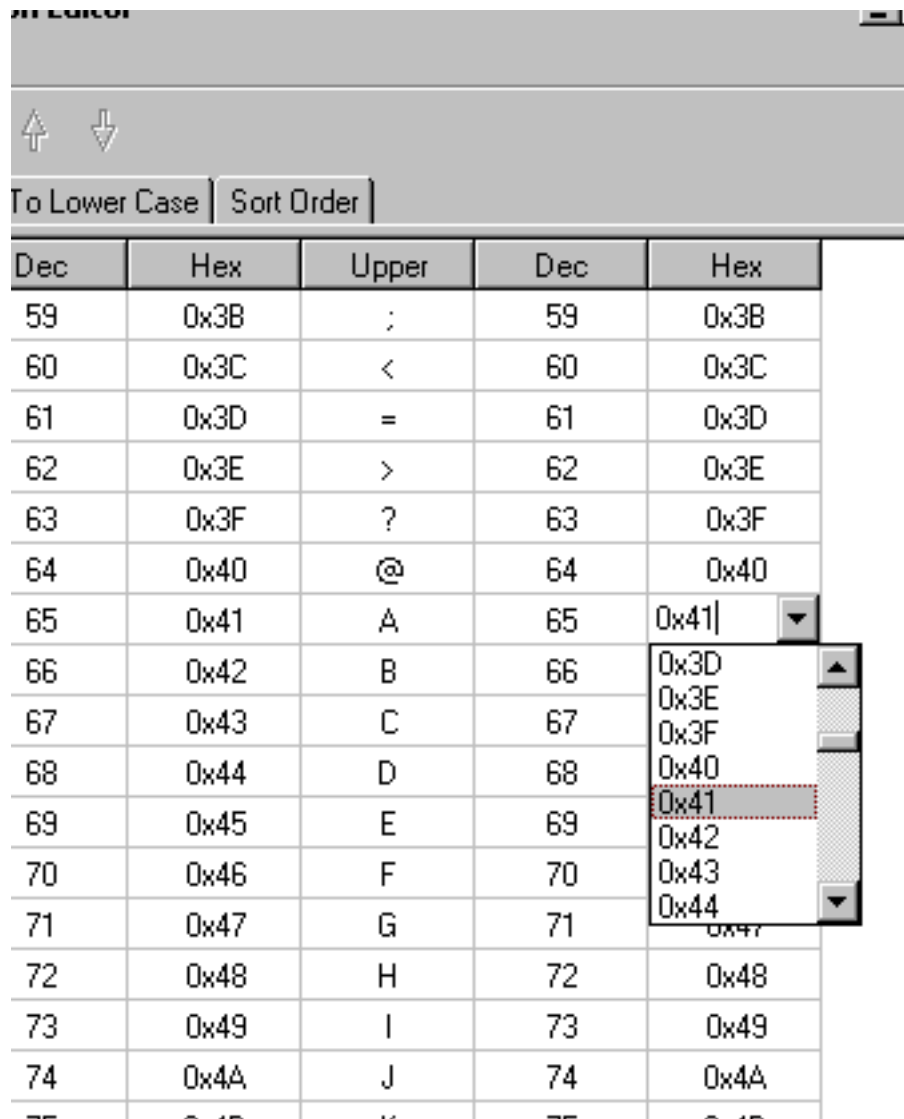
1. Upper case
2. Lower case
3. Collation order

For each of this tables editor shows three panels to edit. On saving editor saves this three tables in the file simultaneously.

Editor allow to edit upper case and lower case rules as different rules and does not check this tables conformance.

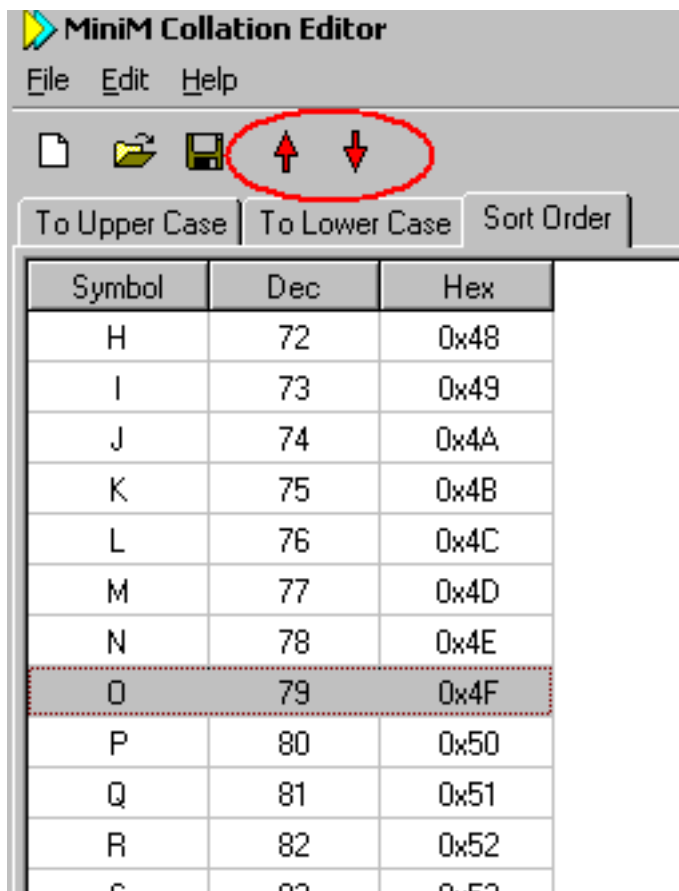
On the panels editor shows three columns - symbol using selected font, decimal and hexadecimal code. To select code or symbol to be in specified position need to click field, activate editor and select new value.





Dec	Hex	Upper	Dec	Hex
59	0x3B	:	59	0x3B
60	0x3C	<	60	0x3C
61	0x3D	=	61	0x3D
62	0x3E	>	62	0x3E
63	0x3F	?	63	0x3F
64	0x40	@	64	0x40
65	0x41	A	65	0x41
66	0x42	B	66	0x42
67	0x43	C	67	0x43
68	0x44	D	68	0x44
69	0x45	E	69	0x45
70	0x46	F	70	0x46
71	0x47	G	71	0x47
72	0x48	H	72	0x48
73	0x49	I	73	0x49
74	0x4A	J	74	0x4A

To change collation order need to switch to "Sort Order" panel, select symbol to move relative others and press on the toolbar buttons with arrows to move one position up or down. Or press Ctrl+Up or Ctrl+Down keystrokes.



On editor start all panels are automatically filled by default values. By default editor defines upper case and lower case for English letters, and upper case and lower case rules for characters greater than 127 does not defined. Collation order by default is a characters code comparison order.

Collation table to be used by the MiniM Database Server instance must be selected in configuration file `minim.ini`, section `Server`, key `Locale`. Value of this key must be a file name without extension and this file must be present in the `/nat` subdirectory. Collation tables are accepted by the server on start and changes of this key value while server still work does not affect to processes.

The table of upper cas rule is used in functions `$zupper` and `$zcvl(str,"U")`. The table of lower case rule is used in functions `$zlower` and `$zcvl(str,"L")`.

Moreover, tables of upper case and lower case rules are used to determine is the character a letter or not - if upper case and lower case of this character are differs from each other, MiniM suppose this character be a letter and use this recognition in the patterns for an "A" patcode.

The collation table is used by the index sorting operator to determine characters following and by subscript comparison to handle subscripts of local, global variables.

To change current collation table for MiniM Database Server instance need to do:

1. Define which globals are use current collation.
2. Export this globals using one of variable length format.
3. Kill this globals from databases.
4. Stop MiniM instance.
5. Specify in file minim.ini new collation file to use.
6. Start MiniM instance again.
7. Import global's data in the same databases.

Collation changes for existing globals can create data phantoms in globals, when one operations can see data but others not. Need to understand that global read, kill and write are a data-driven algorithms. For local variables collation rule changes are safe, because local variables are lost on server stop and start to accept new collation. Restoring from backup requires the same collation rules as it was on backup. Importing globals from block exported data requires the same collation as it was on exporting too.

## 1.10 MiniM license key usage

MiniM Database Server implements license as a special license key file. It is file minim.lic file, placed in the /bin subdirectory of MiniM instance.

License file minim.lic is a text file and use INI file format. Sample of how this file content can appear:

```
[License]
Customer=MiniM Test Group
Count=100
Date=2010.03.31
Key=68DB33AA69A1ADE5078B2F48363
```

File can contain information about who is license owner, how much process allowed to run, expiration date and key signature. License file can contain other additional keys.

If license key does not contain expiration date this key is time unlimited.

If MiniM server cannot find license key, or this file have incorrect data, or date of license is expired, server suppose that have zero licensed processes to run.

Common limitation by process count to work is a licensed processes plu3 3 engineering processes. So, after license expiration or if license file is absent, server allow to run only 3 engeneer processes. In general, 3 additional engeneer processes are intended for special administrative tasks, or to run background processes, or for evaluating software. And, in any case, administrator can use this additional engeneer processes for his discretion and can include this processes into production process set.

Total process limitation for server counts as licensed processes, engeneer processes and process limits have been specified in configuration file *minim.ini*, section *Server*, key *ProcessLimit*. Server uses minimal value. Administrator must check both values on license key replacing. Intended in configuration process limit allow administrator limit memory usage for future MiniM versions with unlimited process counts in license key. On server start counted process number is used to reserve special shared memory area to prevent memory fragmentation and improve server stability.

To change license key administrator must replace license file *minim.lic* in the */bin* subdirectory of MiniM instance to the new content and restart MiniM Database Server. License limits and other configuration options are accepted only on server start.

### **MiniMono difference**

MiniM Embedded Edition does not use any license key, it is free of royalty software.





# Chapter 2

## Devices

### 2.1 Development with TCP device

Before use TCP device developer must choose type of device usage. Device TCP can work in three modes: client, server and concurrent server.

#### **How to use TCP device as a server.**

To use TCP device as a server developer must implement the following steps:

- 1) Open device.
- 2) Make device current and select mode - text or binary and select terminator if need
- 3) Go device to the /ACCEPT operation.  
/medskip After this device is ready to accept incoming tcp/ip connections.  
And after completing the followed command

```
use dev:/ACCEPT
```

following commands executes after incoming tcp/ip connection is established. And following commands a ready to read and write to device and transfer data to and from connected client application.

- 4) Disconnect and close device by the *close* command.

#### **Example how to use TCP as a server**

```
s dev="|TCP|:123" ; 1
o dev ; 2
u dev:/ACCEPT ; 3
r *ch ; 4
```

```
w $c(ch+1)      ; 5
u 0             ; 6
c dev          ; 7
```

Describe this example step-by-step:

- 1) Specified device type TCP and this device must accept incoming connections to tcp port number 123.
- 2) Open device.
- 3) Make device current and goes to /ACCEPT operation to wait incoming connections.
- 4) This command executes after client tcp application connects to port been used. MUMPS code reads character code into *ch* variable.
- 5) Server send character back to client application with code plus 1.
- 6) Make principal device current.
- 7) TCP device closed, other operations with this device are impossible.

For this simple example can be used any telnet client, for example telnet.exe. Run this application and type to connect:

```
open localhost 123
```

To reuse device for other connection it is unneed to close device. Secondary /ACCEPT operation automatically closes connection and goes to wait incoming connections again.

Example above with small changes to handle connections one-by-one:

```
s dev="|TCP|:123"
o dev
f u dev:/ACCEPT r *ch w $c(ch+1)
```

Here in the cycle MUMPS code execute /ACCEPT for TCP device, wait connection, reads one character and send back character with code plus 1. First /ACCEPT operation only wait connection and secondary closes available connection and wait next incoming connection.

More complex servicing example – simple HTTP service. In this example process opens TCP device, waits incoming connections, reads HTTP headers and outputs simple HTTP answer as a very simple HTTP page.

```
socktest
; open socket device
n io="|TCP|:2233"
```

```

; close device if was opened
i $d(~$d(io)) c io
; device was specified as server-side TCP socket on 2233 port
; open it in read-write and text mode
OPEN io USE io:(/MODE="rwt")
continue
; wait incoming connection
; this example does not use timeout for accept
USE io:/ACCEPT
; accepted socket is inside of current device and will be used
; in read and write operation until next accept or closing
d trace("accepted")
; read headers
n line,headers f r line:0.01 q:line="" d
. s headers($i(headers))=line
d trace("headers done")
; dump headers to console
d dump(.headers)
; write answer
d answer
d trace("answer done")
; continue
g continue
answer
w "HTTP/1.0 200 OK", $c(10)
w "Content-Type: text/html", $c(10)
n text="<html><head></head><body>"
s text=text_"Constant HTTP answer.</body></html>"_ $c(10)
w "Content-Length: ", $l(text), $c(10)
w $c(10)
w text
q
trace(str)
; write to principal, not to socket
n saveio=$io
u $p
w "Trace: ", str, !
u saveio
q
dump(var)
q: '$d(var)

```

```

; write to principal, not to socket
n saveio=$io
u $p
n i f i=1:1:var w var(i),!
u saveio
q

```

After running this example in console or in telnet it waits incoming connections from the WEB browser at address

`http://localhost:2233`

and outputs to the operator screen diagnostic messages.

### **How to use TCP device as a concurrent server.**

To implement concurrent server using TCP device it is need to create two parts - parent and child subroutines.

Parent subroutine must create TCP device as a server TCP device, and make /ACCEPT operation to wait incoming connections. After accept connection parent subroutine must run child subroutine as separate job and give accepted socket. Parent part can run the same label or different depending connection characteristics. After running child process parent part is ready to execute /ACCEPT operation again. So, parent part must implement:

- 1) Create TCP device and specify it is server device and specify tcp port to listen.
- 2) Select device mode for parent process.
- 3) Make device current.
- 4) Make /ACCEPT operation and wait incoming connections.
- 5) After executing /ACCEPT parent part is ready to run child job and pass accepted connection.
- 6) Run child job using the *job* command and pass TCP device which have accepted connection.
- 7) After *job* command current device of parent process can be reused without closing and parent subroutine can repeat /ACCEPT operation.
- 8) If it is need, parent subroutine can close TCP device. All child process have been run still in active state and continue to interact with clients.

Child subroutine does not requires any special things to work with device. Child process already have principal device accepted TCP connection, and this device stay in binary mode with an empty terminator. Child process can change device mode or terminator. Child connections still work on the same tcp port as parent connection.

**Example how to use TCP device to implement concurrent server.**

```

; parent server part
srv
n dev="|TCP|:2525"      ; 1
o dev                  ; 2
f q:$d(^STOP) d       ; 3
. u dev:/ACCEPT        ; 4
. j child:(:$io)       ; 5
c dev                  ; 6
q

; child server part
child
u $p:(/MODE="rwt")
w "Child job, #$_$j,!"
r "enter your name: ",name,!
w "Name is: ",name,!
r "Press any key to quit.",name,!
q

```

Describe this example step-by-step.

### Parent process

- 1) Declare device identification string for TCP device as a server and specify tcp port to listen number 2525.
- 2) Open device.
- 3) In cycle while need not stop execute subroutine.
- 4) Device have been opened make current and make /ACCEPT to wait incoming tcp connections.
- 5) Run child process with the *child* label and pass device in concurrent mode (specified as job options (:\$io)).
- 6) After end of cycle device closed and current device is a principal device.

### Child process

This example is a telnet-oriented connection handler and have very simple interface. Child process changes current device mode from binary to text and after this device automatically handle carriage return and line feed symbols. Next child process execute simple interaction operations.

If telnet-client does not show symbols have been entered, it is need to enable local echo mode. For example, Windows Telnet Client can change echo mode by commands after pressing Ctrl+]:

```
set LOCAL_ECHO
```

```
and
```

```
unset LOCAL_ECHO
```

### How to implement TCP client.

To implement TCP client it is required to make the following steps:

1) Create TCP device and specify server name or ip address and port to connect to.

2) Select appropriate device mode, text or binary and terminator.

3) Make device current.

After this device is ready to interact with tcp server specified. TCP connection is made on device opening.

4) Close connection by close command with TCP device.

### Example how to use TCP device as a client.

```
s dev="|TCP|localhost:80" ; 1
o dev:(/MODE="rwt") ; 2
u dev ; 3
w "GET /",! ; 4
f i=1:1:10 r ans(i) ; 5
u 0 ; 6
c dev ; 7
w ; 8
```

Describe this example step-by-step.

1) Declare device identification string for TCP device and specify server *localhost* and tcp port number 80. Here we demonstrate query to local webserver and reading answer.

2) Device is opened in text mode and allow read and write.

3) Make device current. After this all read and write commands receive and send data to webserver.

4) Write to device string "GET /" and line feed. In the HTTP protocol this string mean query to the root page ("/") by the GET method. Line feed is an indicator of the end of http request.

5) In cycle from 1 to 10 lines are read and placed into subscripted local variable *ans*. Device still work in text mode and end of lines are determined by line feed characters received from webserver.

6) Make principal device a current device.

7) Client TCP device closed. Process closes connection to webserver and all internal data related to device are removed.

8) Write to the current device all local variables to see received data.

For example, if we call webserver Apache with default root page, we can get the following answer:

```
ans(1)="<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">"
ans(2)="<HTML>"
ans(3)=" <HEAD>"
ans(4)=" <TITLE>Test Page for Apache Installation</TITLE>"
ans(5)=" </HEAD>"
ans(6)="<!-- Background white, links blue (unvisited),
  navy (visited), red (active) -->"
ans(7)=" <BODY"
ans(8)="  BGCOLOR="#FFFFFF""
ans(9)="  TEXT="#000000""
ans(10)="  LINK="#0000FF""
```

If programmer use binary mode instead text mode, it is required to define read terminator or limit reading by read timeout. Server can send, for example, 100 bytes and client side must correctly determine where data ends.

## 2.2 Development with CON device

### Console escape sequence processor

On output characters to console MiniM process check characters sequence have been sent and search special escape control sequences. Escape sequence starts with special symbol  $\$C(27)$  and terminates by special symbol-terminator, and all sequence from  $\$C(27)$  to terminator ([, ;, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, =, ?) are used as full escape sequence to execute. Delimiter characters splits separate values of escape parameters. Action for escape sequence is depended of escape sequence terminator and other parameters before terminator are escape parameters. In depend of escape sequence parameters can be absent, or can be one, two, or more.

#### Table of symbols for escape sequences

Symbol	Description
Esc	Symbol 2 (decimal code 27, hexadecimal code 1B)

Pn	Number in string representation, decimal digits of number (12, 56, ...)
Pl	Number of display line
Pc	Number of position in the line (column)
Pa	Display attribute

If escape sequence does not contain parameter (parameter is omitted), it is processed with parameter by default. Impossible values of parameters are ignored and unsupported escape sequences are ignored too.

### Control symbols

Code	Description
7	Bell
8	Left by one column
9	Horizontal tabulation, step by 8 columns
10	Line feed (column does not changes)
11	Vertical tabulation (the same as code 10)
12	Form feed (clear screen, caret to the upper left display corner)
13	Carriage return (caret to the begin of line)

### Caret movement commands

Display start (line 1, column 1)	EscH or Esc[H or Esc[f
Caret up (same column)	Esc[PnA or Esc[A or EscA
Caret down (same column)	Esc[PnB or Esc[B or EscB
Caret right	Esc[PnC or Esc[C or EscC
Caret left	Esc[PnD or Esc[D
Positioning (full)	Esc[Pl;PcH or Esc[Pl;Pcf
Save caret position and attributes	Esc[s
Restore caret position and attributes	Esc[u

### Clear screen commands

From caret to the end of line	Esc[0K or Esc[K or EscK
From begin of line to caret	Esc[1K
Entire line	Esc[2K
From caret to the end of screen	Esc[0J or Esc[J or EscJ
From begin of screen to caret	Esc[1J
Entire screen	Esc[2J



**Insert and delete commands**

Insert empty lines	Esc[PnL
Delete lines	Esc[PnM
Delete characters (replace to whitespaces)	Esc[PnX

**Video attributes commands**

Set video attribute	Esc[Pa;Pa;...;Pam
---------------------	-------------------

**Video-attribute codes**

0	all attributes cleared
1	brightness on, high
2	brightness off, normal
30	black text
31	red text
32	green text
33	yellow text
34	blue text
35	magenta text
36	cyan text
37	white text
40	black background
41	red background
42	green background
43	yellow background
44	blue background
45	magenta background
46	cyan background
47	white background

**Show and hide caret**

Hide caret	Esc[?25l
Show caret	Esc[?25h

## 2.3 ATR mnemonic

ATR mnemonic is intended for CON and TNT devices for compatibility with legacy terminal applications and is short version of the SGR mnemonic. The ATR mnemonic is not a part of standards.

In any case the color attribute for /ATR mnemonic is specified as a number greater than 0 and less than 256 by the following table:

Bit number	Value
0, 1, 2	Text color
3	Text brightness
4, 5, 6	Background color
7	Background brightness

So, color attribute can be conveniently specified using hexadecimal numbers where first digit is a background color and second digit is a text color. If it is required to use increased text brightness, need to add to second digit 8, and if need to use increased background brightness, add 8 to first digit.

Colors table:

0	black
1	red
2	green
3	yellow
4	blue
5	magenta
6	cyan
7	white

Examples:

#17	red background, white text
#07	black background, white text
#4B	blue background, bright yellow text
#AF	bright green background, bright white text

Usage examples:

```
W /ATR(#05),"magenta color"
W /ATR(#42),"green text on the blue background"
```

W /ATR(#07),"ordinal white color"



# Chapter 3

## Technical Articles

### 3.1 minim.exe command line switches

MiniM Database Server's executive minim.exe supports several command line switches to control process and pass arguments. Some of switches are documented and can be used in production and some switches are undocumented and can vary from version to version.

Command line switches can be specified in any sequence. If switch have value, this value must follow by switch. For example, switch -h can be specified in any place and switch -x must have value with specification what must be executed and this value must directly follows by the switch.

#### **Switch -std.**

This switch is used to directly specify that new MiniM process must be run with STD default device. In this case process use standard input-output channels stdin and stdout. Input and output can be redirected to and from disk files or other applicaitons standard stdin and stdout channels. If minim.exe process have been run with redirection, process automatically determine needs to use STD device event if -std switch does not specified.

If process been run in console mode and can accept user input from keyboard, this process automatically use CON device as default device. If process have been run interactively but must use STD device by default, switch -std do this. And in this case process does not use console control and use keyboard input and output to screen as other simple applications. For example, this mode can not highlight symbols on the screen.

#### **Switch -x**

Switch `-x` is intended to specify what commands must be executed on process start. If following command line option starts with special symbol `@`, next option part is used as a file name to get lines of code to execute. Otherwise following option is a command sequence need to execute. Lines in file can be delimited by empty lines and must be created like entered from the keyboard, without labels and dot syntax. For example:

```
minim.exe -x "w $zv,!"
```

this command runs MiniM process and specify to execute MUMPS command

```
w $zv,!
```

Other example:

```
minim.exe -x @script.m
```

this command runs MiniM process and specify to execute lines of code from `script.m` file line-by-line.

### Switch `-h`

Switch `-h` specify that MiniM process must terminates after all commands from command options been executed. Process terminates by executing the *halt* command. Process can execute command *halt* if this command been specified in command options or in any code been called too.

### Switch `-nspace`

Switch `-nspace` specifies current database name for new process, name must follows switch keyword. Example:

```
minim.exe -x "d label^routine" -nspace USER
```

If current database for new process does not specified, process use as current database default database for new process, specified in the `minim.ini`, section `[Process]`, key `Namespace`.

### Switch `-ignore`

Switch `-ignore` is used to specify this switch and the following command options does not need to analyze. If this switch is present, process stops to analyze followed options.

Switch `-ignore` is intended to use for run process from other environment which automatically append command options, for example webserver Apache if `minim.exe` is used as direct cgi-module.

### Other switches

Other switches are internal and undocumented or unsupported. If MiniM process detects unsupported switch in command line options, process ignores this switch but make special record into `minim.log` file. Unsupported switches does not affect to MiniM process, but MiniM count this switches as errors and record to `minim.log` file is a warning. This may be misprint or an error.

### Input-output redirection

To redirect input-output MiniM supports redirection using symbols `<`, `>`, and `|`. If MiniM process detect redirection, process starts with STD device by default and if redirection been terminated, process terminates too. If process been run with redirection, device STD does not output prompt as current satabase name and `>` symbol and does not make line feed after each line of command execute. Process suppose that output formatting have been specified directly. MiniM process have not any output to `stderr` standard channel.

Examples how redirection can be used:

- 1) `minim.exe < cmds.m`
- 2) `minim.exe < cmds.m > out.txt`
- 3) `minim.exe < cmds.m >> out.txt`
- 4) `echo w $zv,! | minim.exe`
- 5) `echo w $zv,! | minim.exe > out.txt`

Here first example runs MiniM process with input lines of command from `cmds.m` file. All lines are compiled and executed sequentially, line-by-line. After last line execution process terminates.

Second example runs the same as first but redirects output to `out.txt` file unlike to screen as in first example. After all commands been executed all output is stored in the `out.txt` file.

Third example is the same as first and second, but output does not replace `out.txt` file content if this file already exists, and append output to existing content.

Fourth example runs `echo` Windows command, which outputs symbols `w $zv,!` with redirection to input channel of MiniM process. All `echo` argument to `|` symbol is placed into input channel of MiniM process.

Fifth example is the same as fourth example but redirects output to the out.txt file, not to screen.

Input-output redirection is intended to use MiniM Database Server in different cases and integrate with different other applications using open interfaces.

## 3.2 ZDLL module development

MiniM Database Server implements calls to external dynamic link libraries. This library can be written using any compiler which can build Windows 32 bit dll. This calls are named as ZDLL and this special dynamic link libraries are ZDLL modules. MiniM allow calls and data transfer from MUMPS to external ZDLL module as such as from ZDLL module to internal MiniM environment.

Programming interface on the MiniM side is made by special function *\$zdll()* and this function is described in the MiniM Language Guide.

Now application programming interface is present in the C/C++ interface and oriented for widely used compilers. And this interface is intended in the form and calling conventions which allow to use any other Windows - oriented development tools. In this case programmer must use special conventions of selected language and compiler to create dll.

Common principles of ZDLL module are the standard principles to work with special dll extensions. System function *\$zdll()* searches library file, loads and initializes module in own memory space and calls special function to get available exported functions ready to call from MiniM environment. This special function must return array with exported functions definitions. Array must be ended with zeroed record. ZDLL can implement one or more functions and must define this functions in one definition array. Function which returns exported functions definitions is called only once and function must return array in memory available at all time ZDLL module is loaded. This memory can be static or dynamic and in second case ZDLL module must free used memory independently on library unloading.

One definition of exported function consists of two fields - pointer to function who must be called with fixed prototype and pointer to ASCII string with function name for late binding. This string can contain any characters and MiniM search functions by this names case insensitive.

Each exported function must be provided by pointer and must accept four arguments. It is pointer to special structure with callback functions and



pointer to buffer for function result. Function *\$zdll()* before call initializes data in the return buffer as an empty string and in any case *\$zdll()* function return result, at least empty string. Third parameter accepts real argument count passed by *\$zdll()* function to ZDLL module and fourth parameter is an open array of values for passed parameters. So, one function can accept variable length parameters.

Exported function of ZDLL module can call internal MiniM environment for 1) evaluate MUMPS expression, 2) execute commands of MUMPS language like *xecute* command and 3) call functions and subroutines with direct arguments values passing. First and second call required string construction on MUMPS language with doubling quotes as need and third call does not, values are passed as is.

All data transfer between MiniM environment and ZDLL module are made using special unified structure *MINIM\_STR*, defined in *zdll.h* header file. This structure contains two fields - *len* as tag or data length and data bytes array. If value of *len* is greater than or equal zero, this mean string passing with this length and data must contain bytes of string. Otherwise *len* field contains special tag indicating type of number been passed (*MT\_INT32* for int32 integers, *MT\_INT64* for int64 integers or *MT\_DOUBLE* for doubles) and in *data* field must be binary number representation in Intel byte encoding.

To convert from one type to another, read and write non-string data MiniM give several special functions *GetDouble*, *GetInt32*, *GetInt64*, *GetStr*, *SetDouble*, *SetInt32*, *SetInt64*. To create string data ZDLL function must allocate buffer for *MINIM\_STR* structure, specify data length and fill array of characters. All other bytes over specified *len* are ignored and string can contain any characters including zeroes and nonprintable. Convert functions returns a result independently of the passed format, for example, *GetInt32* return an integer even been passed string or double and *GetStr* return string representation even been passed one of the number form.

Callback function *ErrStr* returns current value of the *\$error* system variable.

MiniM installer installs three examples how to write external ZDLL module. First example show how to accept several arguments from MiniM environment and create return value. Second example show how to call from ZDLL module to MiniM environment to evaluate MUMPS expression. And third example show how to execute line of MUMPS commands and change current state of MiniM environment. Next will be described examples and function prototypes step-by-step. Assumed that programmer have

base knowledge about functions, pointers to function, structures, pointers to structure, arrays of structure, export functions and dll linkage.

```
int WINAPI DllEntryPoint(HINSTANCE hinst,
    unsigned long reason, void* lpReserved)
```

This function is called by Windows environment on dll loading and unloading and in this function programmer must do any actions about initialization and deinitialization of ZDLL module. In this function context MiniM environment is inaccessible.

```
#ifdef __cplusplus
extern "C"
#endif
__declspec( dllexport ) ZDLLFUNC* __stdcall ZDLL ( void)
{
    return functions;
};
```

Here is exported function with predefined prototype and name. This function will be searched by MiniM process to get available functions to execute. ZDLL module can export any other functions too, but all other functions are unused by MiniM. If ZDLL module does not export this function, this ZDLL module cannot work with MiniM. In example function return pointer to array or records with supported function definitions. An example uses static array of records placement. Also ZDLL module can use dynamic memory allocation, but also must free this memory. Memory with array of definition records must be available all time ZDLL module is loaded.

```
ZDLLFUNC functions[] =
{
    { concat, "concat"},
    { NULL, NULL }
};
```

Here is described array or records with first element to define pointer to function and name of function to call from MiniM environment. Second array element is an empty record to define end of functions definition. This array can define any number of functions but last element must be an empty record.

```
int __stdcall concat( ZDLLCB* cbfunc, MINIM_STR* result,
    int argc, MINIM_STR** argv)
```

Pointer to this function is used in ZDLL function definition and can be called from MiniM environment by name "concat". This name is specified in definition array. Name as string and name as real function name in code can differs, MiniM use only pointer to function. Function accept four formal parameters and on call from *\$zdll()* function get real arguments passed count and array of arguments values.

```
MINIM_STR a;
MINIM_STR b;
```

Function use temporary structures to convert data into strings. MiniM does not guarantee format of values been passed - string or a number and ZDLL function must cast data to need format to use.

```
cbfunc->GetStr( argv[ 0], &a);
cbfunc->GetStr( argv[ 1], &b);
```

Here function calls MiniM environment to convert data into string format. If data has been numbers, MiniM creates canonical string representation, if data has been strings, MiniM hold this values as is.

```
if( a.len + b.len > MINIM_STR_MAX)
{
    return 1;
}
```

Here function check that result does not exceed maximum allowed length. If this occurs, function return nonzero result and in this case MiniM function *\$zdll()* generates an error <FUNCTION>. Function of ZDLL module cannot access memory over allocated. If dll module crash MiniM process, locking objects, shared data and transactions will not be lost, all internal data will be cleared by guardian process, but other process working and process state restoring after crashing is impossible.

```
memcpy( result->data, a.data, a.len);
memcpy( result->data + a.len, b.data, b.len);
```

Here function create result as concatenation of data from source formal parameters, copy first data and next second data.

```
result->len = a.len + b.len;
```

Function declare how much bytes is used by result.

```
return 0;
```

Fuction return zero to indicate all done and function finished successfully.

So, function from first example can be called from MiniM environment as written in file exam1.rsa:

```
$zdll("call","exam1.dll","concat",1234,"abcd")
```

Here example demonstrate that MiniM does not change passed values format and data can be string and numbers and if function requires special format, function must convert data. On the other side, programmer can decide need conversion or not.

Next is described second example without element been described above, such as definition export.

```
int __stdcall getdate( ZDLLCB* cbfunc, MINIM_STR* result,
    int argc, MINIM_STR** argv)
```

This is function, really does not used any passed arguments.

```
MINIM_STR expr;
```

Function use temporary data to call MiniM environment.

```
expr.len = sprintf( expr.data, "$zd($h,13)");
```

Function create MUMPS expression to evaluate later and create bot byte sequence and byte sequence length.

```
cbfunc->Eval( &expr, result);
```

Function call MiniM environment to evaluate expression and get result. Here function use the *result* variable directly to return evaluated result as *\$zdll()* function return.

```
return 0;
```

Function return zero value to indicate successful execution. In real applications function must check *ErrStr* function return and check that was not any errors in MiniM environment while expression evaluates. It is recommended to check MiniM process state after each call MiniM environment, or to check return values of functions *Eval* and *Execute*. If this return value is ZDLL\_CALLBACK\_DONE, functions complete successfully, otherwise functions return predefined error code as ZDLL\_CALLBACK\_SYNTAX, ZDLL\_CALLBACK\_ERROR or ZDLL\_CALLBACK\_UNDEFINED. If arguments passed been formed incorrectly, functions return an error code as ZDLL\_CALLBACK\_PARAMETERS.

Routine in file exam3.rsa contains example how to call this ZDLL function:

```
$zdll("call","exam2.dll","getdate")
```

Here *\$zdll()* function does not pass any additional arguments.

Next is described third example without parts been described above such as function definition.

```
int __stdcall set( ZDLLCB* cbfunc, MINIM_STR* result,
  int argc, MINIM_STR** argv)
```

This function to call from MiniM environment really does not use any arguments passed.

```
MINIM_STR str;
```

Function use temporary data to create line of code to execute.

```
str.len = sprintf( str.data, "s a(1)=1");
```

Function create line of code to execute as byte sequence and mark total bytes used length. Function form bytes sequence in *data* field and total length in *len* field.

```
cbfunc->Execute( &str);
```

String created above is passed to execute in MiniM environment as a line of MUMPS commands.

Note for ZDLL module programmers: while MiniM environment call functions from ZDLL module, the *result* points to the stacked structure and inside recursive calls MiniM -> ZDLL -> MiniM -> ZDLL programmers can use data in *result* variable safely.

While MiniM execute callback functions *Execute*, MiniM creates new stack level like for a *ecute* command. And all local variables been created by the *new* command, are lost on return from *Execute* and stack level is restored.

All cakkback functions *UserFunc*, *UserDo*, *ReadLocal*, *WriteLocal*, *ReadGlobal*, *WriteGlobal*, *KillLocal*, *KillGlobal*, which can be named as low-level callback functions, does not creates new stack level to execute. And, this functions return error code defined as:

```
#define ZDLL_CALLBACK_DONE          ( 0 )
#define ZDLL_CALLBACK_SYNTAX       ( 1 )
#define ZDLL_CALLBACK_PARAMETERS   ( 2 )
#define ZDLL_CALLBACK_ARGC         ( 3 )
#define ZDLL_CALLBACK_UNDEFINED    ( 4 )
#define ZDLL_CALLBACK_ERROR        ( 5 )
#define ZDLL_CALLBACK_HALT         ( 6 )
```

Low-level functions table:

UserFunc	Evaluate user function written in MUMPS and return value.
UserDo	Call subroutine written in MUMPS and does not return value.
ReadLocal	Reads local variable value.
WriteLocal	Writes or creates local variable value.
ReadGlobal	Reads global variable value.
WriteGlobal	Writes or creates global variable value.

KillLocal	Removes local variable with subscripts.
KillGlobal	Removes global variable with subscripts.
SetTEST	Return current and sets new value of system variable <i>\$test</i>
OrderLocal	Return \$order function result for local variable
OrderGlobal	Return \$order function result for global variable
IncLocal	Increment local variable by 1 and return result
IncGlobal	Increment global variable by 1 and return result
DataLocal	Return \$data function result for local variable
DataGlobal	Return \$data function result for global variable

Return values table:

ZDLL\_CALLBACK\_DONE - Action done successfully.

ZDLL\_CALLBACK\_SYNTAX - One or more arguments have a syntax error.

ZDLL\_CALLBACK\_PARAMETERS - One or more passed arguments have impossible or unsupported values.

ZDLL\_CALLBACK\_ARGC - Have been specified impossible arguments or subscripts count.

ZDLL\_CALLBACK\_UNDEFINED - On read been detected that variable have undefined value.

ZDLL\_CALLBACK\_ERROR - Other error been detected.

ZDLL\_CALLBACK\_HALT - Command HALT was reached in MiniMono context, application must exit and unload library MiniMono.dll

To call user function using *UserFunc* string with function name must starts with characters "\$\$". For example, to call user defined function *\$\$func^cbfunc* with first argument as 5555, sample code can be as follows:

```
char* name = "$$func^cbfunc";
MINIM_STR result;
MINIM_STR arg;
cbfunc->SetInt32( 5555, &arg);
MINIM_STR* argv[ 1];
argv[ 0] = &arg;
cbfunc->UserFunc( name, 1, argv, &result);
```

To call user defined subroutine using *UserDo* string with subroutine name must contain only subroutine name. For functions *UserFunc* and *UserDo* name of routine can be omitted, in this case MiniM environment use current routine if one is present in current execution context, for example:

```

char* name = "func";
MINIM_STR arg;
cbfunc->SetInt32( 5555, &arg);
MINIM_STR* argv[ 1];
argv[ 0] = &arg;
cbfunc->UserDo( name, 1, argv);

```

Low level callback functions does not require double quoting of subscripts and arguments unlike of *Eval* and *Execute* functions to create correct MUMPS syntax. Low-level callback functions use arguments been passes as is. If been passed string as subscript value, MiniM environment check this value can be a number and use internal conversion as need.

Functions *ReadGlobal*, *WriteGlobal* and *KillGlobal* does not required database name. If instead of database name been passed NULL value or an empty string, MiniM process use current database. The circumflex character (^) does not required in the global name.

If MiniM environment detect that *ReadLocal* or *ReadGlobal* functions calls to variable with undefined value, this functions returns

```
ZDLL\_CALLBACK\_UNDEFINED
```

error code and variable *result* does not contain any value and does not changed. Otherwise MiniM environment writes value been read into *result* and return error code ZDLL\_CALLBACK\_DONE. MiniM return value in the same format (string or a number) such as this value been stored in a variable. To convert value into need format programmer must use conversion functions *GetDouble*, *GetInt32*, *GetInt64*, *GetStr*.

Example how to read and write global variable *^GlobalName("abc")*:

```

// write variable
MINIM_STR value;
value.len = sprintf( value.data, "%s", "abcdef");
MINIM_STR index;
index.len = sprintf( index.data, "%s", "abc");
MINIM_STR* argv[ 1];
argv[ 0] = &index;
char* name = "GlobalName";
int ret = cbfunc->WriteGlobal( name,
    NULL, 1, argv, &value);

```



```
// read variable
ret = cbfunc->ReadGlobal( name,
    NULL, 1, argv, &value);
```

If process changes value of system variable *\$test* using function *SetTEST*, value of *svn\_test\_value* is compared with zero. If it is zero, MiniM environment sets *\$test* to 0, otherwise sets to 1.

### 3.3 ZDEVICE module development

MiniM Database Server implements extending internal built-in device types with user-defined device types in external dynamic link libraries. In general, this external defined devices are named as z-devices or ZDEVICE module.

To create z-device module programmer must use any compiler to create Win32 dll and implement z-device as set of functions with special prototypes. MiniM installer have some examples created with C++.

DLL module must export one function with predefined prototype and with name *GetZDevice*. This function must return to MiniM environment pointer to structure with pointers to functions as handlers of device events. All function prototypes are defined in the header file *zdevice.h*.

After z-device been created, MiniM process can use this device as such as other or as built-in MiniM device. To use z-device it is required to place dll with z-device implementation in the */bin* subdirectory of MiniM installation. Device type must be the dll name without extension and dll name have to starts with the "Z" character. For example, if programmer plans to use device

```
s devname="|ZDEV1|name"
```

the MiniM process will search the *zdev1.dll* file in the */bin* subdirectory.

Z-device module must implement at least one function, the open device event handler. Other event handlers can be implemented in discretion of z-device developer and are optional. For example, device can not implement system variables *\$X* and *\$Y* support. All unimplemented event handlers must have NULL pointers.

After this special dll file is placed into the */bin* subdirectory, all MiniM processes can open and use this device, it is not required any special registration. Dynamic link library is loaded into process's address space on first

device with this type been open and is freed if last device with this type is closed. And, if process use several devices of this type simultaneously, dll module is unloaded only after all this devices be closed.

Dynamic link library for z-device must be created very like as ZDLL module and can implement both z-device and ZDLL interfaces. If application requires very frequently open and close z-device, application can previously load the library into memory by call to `$zdll("load")` function before using and call to `$zdll("unload")` function after using, if this dll implement both ZDLL and ZDEVICE interfaces. Between this calls library stay loaded and device opening and closing can be faster. If dll does not require to implement any ZDLL functionality, dll can export an empty function definition array.

Functions of z-device event handlers accept at least one parameter with callback functions and accepts data in the same conventions as a ZDLL modules do. While handler function still active, it can call MiniM environment and convert data.

### Interface definition

Indetface of z-device is defined by one structure with pointers to functions handlers or events occured with the device been implemented. Functions accepts al least two parameters, it is pointer to structure with callback functions to call MiniM environment

ZDLLCB\* cbfunc

and current device instance context, which creates the open device handler

ZDEVICEHANDLE hDevice

Current device context is created on device open event. Value of the *hDevice* must not be NULL even if the device events does not use current device context. If open habdler return a NULL value, MiniM process suppose device opening failed and device have not been opened. MiniM process does not use this value in any case except passing to event handlers. This value is required to identify current device instance from other possible device instances can be created simultaneously. ZDEVICE interface is oriented to support several device instances of one device type.

Function habdlers of device events must return an error code to indicate success. If return code is equal to ZDEV\_NO\_ERROR, MiniM suppose event handler executes successfully and process continues execution. If return code is equal one of the following error code, MiniM process will generate an appropriate error:

<b>Error code</b>	<b>Error to generate</b>
ZDEV_ERR_READ	<READ>
ZDEV_ERR_WRITE	<WRITE>
ZDEV_ERR_ENDOFFILE	<ENDOFFILE>
ZDEV_ERR_DEVICE	<DEVICE>

If handler return any other error code, MiniM process use this code to construct the error text and starts error text with "ZDEVICE" characters and with following code, for example if event handler return error code as 456, MiniM process will generate an error <ZDEVICE456>.

All event handlers except device opening handler are optional. If ZDEVICE developer suppose that this ZDEVICE does not implement any functionality, this event handler must be NULL pointer. And if MiniM process calls unimplemented event, process do default action for this event. For example, if write handler *WriteStr* is not implemented, all write string events lost data and if ZDEVICE does not implement reading value of system variable *\$X* for this device, process return by default value 0.

Functions for *Open*, *Close* and *Use* events accepts options if ones have been specified in the MUMPS code. Options are passed as array of pointers to names and values of options and as count of pairs been passed. This array contains names and values strictly sequantially, as a name with the followed value. If in selected position option has been skipped, this pair is a two empty pointers, name and value. If option have been specified without name, name pointer is a null pointer with not-null value. If option have been specified as name only, name is not null and value is null. Note that counter specify a pairs count and is doubled count of pointers been passed. The name and the value are passed in internal MiniM representation as this value have been evaluated and ZDEVICE developer must convert data into need format to use. ZDEVICE developer can independently define any options meaning, using option by position or by name and what options values does.

If device have assigned routine to handle mnemonics, menonics are handled by MiniM process. By default some of device types can have mnemonics assigned using *minim.ini* configuration file. And z-devices have not any mnemonic routine assigned by default.

The *Open* event handler function called by *open* command to open one more device with this type. Function accepts MiniM environment context with callback functions, options and new device name have been specified in the *open* command. Device name is a string which follows by the device type. For example, if MUMPS code opens device "|ZDEV1|abcd", event

handler got string "abcd" (zero terminated) as a device name. Next device identification is made by full name including device name and device type.

ZDEVICE developer can use device name been passed at his own discretion. In any MiniM process all devices are created separately and belongs to process only, not to server or to other process. ZDEVICE developer must allow creation device for other processes with the same name as different objects.

MiniM environment calls *Close* event handler id MUMPS code closes this device. After closing MiniM process free all internal data about this device and free this ZDEVICE module if no more this devices still opened.

MiniM environment calls *Use* event handler on the *use* command and pass specified in MUMPS code options. At this moment this device instance can be current and *stands* a current device after *use* command handled and *Use* handler return control.

MiniM environment calls *ReadStr* event handler on the *read* command in read string form. Event handler accept read length limit and timeout if ones been specified. If read length limit or timeout has not been specified, MiniM environment pass -1 values. Timeout is passed as integer and contains milliseconds but MUMPS command specify timeout in seconds, this recalculation is done automatically by MiniM to conform common MiniM timeouts rules. *ReadStr* event handler must read data from actual data source and place result into parameter

`MINIM_STR* str`

Format of this value (string or number) is defined by ZDEVICE developer and is used by MiniM as is.

The *ReadChar* event handler is called by MiniM environment to handle *read* command in read one character form. This event handler accepts timeout to read in milliseconds or value -1 if timeout does not been specified in MUMPS code. This event handler must read one byte from actual data source and place the code of character into parameter

`int* result`

MUMPS language use conventions that read one character command must return code -1 on unsuccessful reading or if timeout have been expired. On successful read event handler must return the code between 0 and 255 inclusively.

In the special case that ZDEVICE developer suppose that *read* command for read one character for this device type should return value over standard interval of codes, this ZDEVICE developer should specify this feature in documentation for this new device.

Event handlers for *ReadStr* and *ReadChar* events must detect reading outside available data and return error code ZDEV\_ERR\_ENDOFFILE in this case. And MiniM environment in this case analyze current process context and can ignore this return code and MUMPS code must check *\$zeof* variable or process generate an error <ENDOFFILE> and MUMPS code must handle this error.

ZDEVICE developer must understand that *read* command can write to the device. This cases are *read* command in the following forms:

```
read "any string"
read ?expr
read !
read #
```

and to implement this forms ZDEVICE developer must implement appropriate write handlers, otherwise by default MiniM environment lost this data. Note that read command with number as an argument is deprecated by the MUMPS standard, for example:

```
USER>r 123
```

```
<SYNTAX> :READ: *r 123
```

MiniM environment calls *WriteStr* event handler to handle write command in the write string form. MiniM pass string to write in the parameter

```
MINIM_STR* str
```

and data are passed in MiniM internal format (string or a number) as is and ZDEVICE developer must convert data to need format.

MiniM environment calls the *WriteChar* event handler on the *write* command in the write one character form. Here can be passed negative value too. What does it mean and how to handle negative values ZDEVICE developer must suppose in dependent of the device type.

MiniM environment calls the *WriteNL* event handler to handle the write *New Line* (write !) command. What does it mean for this device ZDEVICE developer must suppose in dependent of the device type.

MiniM environment calls the *WriteFF* event handler to handle the write *Form Feed* (write #) command. What does it mean for this device ZDEVICE developer must suppose in dependent of the device type.

MiniM environment calls the *WriteTab* event handler to handle the write tabulation (write ?expr) command. What does it mean for this device ZDEVICE developer must suppose in dependent of the device type. This event handler accepts parameter with value of evaluated expression *expr*.

The *GetX* and *GetY* event handlers must be implemented if this z-device must support system variables *\$X* and *\$Y* for read. What does it mean for this device ZDEVICE developer must suppose in dependent of the device type.

The *SetX* and *SetY* event handlers must be implemented if this z-device must support system variables *\$X* and *\$Y* for assignment. What does it mean for this device ZDEVICE developer must suppose in dependent of the device type.

The *GetKEY* and *SetKEY* event handlers must be implemented if this device must support system variable *\$key* for reading and for assignment. What does it mean for this device ZDEVICE developer must suppose in dependent of the device type. In the most cases value of *\$key* contains last characters have been read and assignment does nothing except storing data until next *read* command executes.

The *GetZEOF* event handler must be implemented if z-device must support system variable *\$zeof* to indicate end of data been reached. This handler return must be coordinated with *ReadStr* and *ReadChar* event handlers from the point of view of MUMPS developers.

The *GetZA* and *GetZB* event handlers must be implemented if this device must support system variables *\$za* and *\$zb*. What does it mean for this device ZDEVICE developer must suppose in dependent of the device type.

Event handlers *GetKEY*, *GetZA* and *GetZB* are called by MiniM environment on each call to system variables *\$key*, *\$za* and *\$zb* from MUMPS code. MiniM process does not cache this values and return values must be determined and calculated on each call been made.

### **ZDEV1 example**

MiniM installer installs as an example how to write read handlers for z-device example ZDEV1 in the *zdevice/example1* subdirectory. This device generates a random byte sequence in internal data buffer and later return strings and characters in read handlers. This device is a read only device.

As an example how to use *use* command options this demo device support one option with the */POS* name for direct positioning in the currently used buffer, for example

```
use dev:/POS=123
```

and this code can reposition current internal read pointer in the buffer and MUMPS code can reread data again.

For illustration how this device can be used is intended routine ZDEV1, which contains actions for read one string (*readstr*), read one string with length limit (*readlimit*), read all data with <ENDOFFILE> error generation (*readtofail*), read all data with *\$zeof* handling (*readtoeof*) and repeatable read after repositioning (*doubleread*).

```
run ; k d run^ZDEV1 w
  d readstr
  d readlimit
  d readtofail
  d readtoeof
  d doubleread
  q
readstr
  ; read one string
  n dev="|ZDEV1|123",str
  o dev u dev r str u $p c dev
  w "str is ",$l(str)," length",!
  q
readlimit
  ; read one limited string
  n dev="|ZDEV1|123",str,limit=1024
  o dev u dev r str#limit u $p c dev
  w "str is ",$l(str)," length",!
  q
readtofail
  ; read all to <ENDOFFILE> error
  n $set="g err"
  n dev="|ZDEV1|123",str
  o dev u dev f r str
  u $p c dev
  q
err
```

```

u $p
w "reached ",$ze," error",!
s $ec=""
c dev
q
readtoeof
; read all to $zeof indicator
n dev="|ZDEV1|123",str,zeof=$v("proc",5,0)
o dev u dev f r str q:$zeof
u $p c dev
i $v("proc",5,zeof)
w "reached $zeof",!
q
doubleread
; read one string, rewind and read again
n dev="|ZDEV1|123",str1,str2
o dev u dev r str1 u dev:/POS=0 r str2
u $p c dev
w "strings compared: ",(str1=str2),!
q

```

Output of this sample routine::

```

USER>k d run^ZDEV1 w
str is 32767 length
str is 1024 length
reached <ENDOFFILE> :readtofail+4^ZDEV1: error
reached $zeof
strings compared: 1

```

### ZDEV2 example

As a second z-device example MiniM installer installs ZDEV2 example into subdirectory *zdevice/example2*. This device is a write-only oriented device and outputs all data into internal debugging port of Windows using WinAPI function *OutputDebugString*. If debugging monitor is run, developer can see output information about how event handlers work and see handlers parameters.

Using this device, ZDEVICE developer can check what event handlers are called on the special read command forms.



```

run ; k d run^ZDEV2 w
n dev="|ZDEV2|123"
o dev u dev
w 67
w *67
w !
w #
w ?67
u $p
c dev
q

```

Exmample what can be displayed in the Windows debugging monitor db-mon.exe:

```

600: ZDEV2:WriteStr: 67
600: ZDEV2:WriteChar: C (67)
600: ZDEV2:WriteNL
600: ZDEV2:WriteFF
600: ZDEV2:WriteTab 67

```

### 3.4 User-defined z-functions

MiniM Database Server implements user-defined z-functions and user-defined z-variables. Both this thing next described as z-functions, because user variable call in MiniM is equal to user function call with zero arguments.

User-defined z-function is a function with name starts with special character "\$" with followed character "Z" or "z" and any English letters or digits. If arter name follows parenthesis, it is user-defined function, otherwise it is user-defined variable. User-defined z-functions names are used case-insensitive.

Z-function implementation must be written in MUMPS language and placed into routine with any name starts with characters "%ZFUNC" case sensitive. After characters "%ZFUNC" may follows any English letters or digits up to 31 symbols in total.

User-defined z-functions calls without routine specification, for example:

```

w $zmyfunc(123,456)
s a=$zmyext(.b)

```

User-defined z-functions cannot use internal MiniM z-functions in this case MiniM call internal functions instead user-defined code.

In implementation routine %ZFUNCXXX for user-defined z-functions must be present label with appropriate formal parameters. Label name must be in upper case only. MiniM process on user-defined z-function call search routine which starts with %ZFUNC in order index sorting. If process find routine with need label in upper case, process use this label to call function. If database contains two or more routines with this label, process use only first found.

Example. Let it be a routine %ZFUNC001 with content:

```
ZZVAR
  q 123
ZZFUNC(param)
  q param
```

then this can be used as user-defined z-variable ZZVAR and user-defined z-function ZZFUNC with one argument:

```
USER>w $zzVar
123
USER>w $zzFunc(789)
789
```

If user-defined function is used, process use only compiled bytecode. If compiled bytecode is absent even if source code is present, process generates an error <NOZROUTINE>. If process found one or more %ZFUNC routines but label was not found, process generate an error <NOZROUTINE> too.

MiniM use arguments passing to user-defined z-functions in the same manner as for other user-defined function and may be used formal parameters with default values and variable-length parameters.

### 3.5 User-defined z-commands

MiniM Database Server implements extending by user-defined commands. This command names must starts with the "Z" or "z" character and have different name than implemented by MiniM as built-in commands. User-defined commands must be implemented as a subroutines in routines with names starts with "%ZCMD" characters.

For user-defined command can be used argumentless form and argumented form and argumented command can be implemented with one argument only. If MUMPS code call this command with several arguments commadelimited, MiniM environment call one-argument form for each argument been specified sequentially in left-to-right order.

User-defined command can be used with postconditional expression. In this case postconditional expression evaluates ones before command execution independently of arguments count. If postconditional expression evaluates as 0, all arguments are ignored and control is passed to the next command available.

Z-command implementation must be defined using MUMPS language in routine with naem starts with "%ZCMD" characters with optional followed English letters or digits. Common routine name must be up to 31 characters length and conform common routine name rules.

Each used user-defined z-command must have appropriate label in the "%ZCMDXXX" routine in upper case. On execution MiniM process searches this routine inside available in subscripts collation order. If process does not find any of "%ZCMDXXX" routine, process generate an error <NOZROUTINE> and if in this routines does not found need label, process generates an error <NOZROUTINE> too.

User-defined z-functions cannot have the same names as MiniM built-in commands. In this case MiniM compiler use this built-in functions and built-in functions cannot be redefined.

Label name must be the same as a command name and be in upper case only. For example, if command is *zmycmd*, label must be *ZMYCMD*, if command is *zzshow*, label must be *ZZSHOW*. User-defined commands can be used case-insensitive, but labels must be in upper case.

Example. Let it be a %ZCMD001 routine with the text:

```
ZCMD0
  w 123,!
ZCMD1(param)
  w param,!
```

and in this case can be used command *zcmd0* without argumants and *zcmd1* with one argument. For *zcmd1* command can be used several arguments and in this case MiniM calls subroutine *ZCMD1 ^%ZCMD001* for each argument specified.

User-defined command can accept only one formal parameter and by value only.

To use user-defined commands and pass arguments can be specified any expressions, MiniM process evaluates expressions and pass evaluated values to subroutine.

### 3.6 Processes accounts

To configure correctly MiniM Database Server and as a consequence, for correct functioning, it is required to understand the server policies related to server processes.

MiniM Database Server includes several processes: service (mnmsvc.exe), journaling daemon (minimjd.exe), write daemon (minimwd.exe), database expand daemon (minimed.exe), optional controller to start and stop server (minimti.exe), and the work processes to implement MUMPS jobs (minim.exe). MiniM server controller (minimti.exe) is not mandatory for server and used to run optional utilities by operator.

Main server account for administering is a service account. Service runs child daemon processes, handle telnet incoming connections and runs child process and guard child processes. All child processes which are run by the service, are under the same account. And, if minim.exe process runs child process by the *job* command, child process is under the same account. Also minim.exe processes can be run directly as console or in the standard batch mode. And in this case processes are run under the current user account.

All child processes which are run by the service on the server start (database transactions check and autostart execute) are run under service account too.

In the most cases process accounts are tolerant for MiniM Database Server still work properly. But in special cases this may require understand accounts details: 1) network resources usage and 2) server functioning on current user logoff.

If user logoff, this user account ends and all processes was run under this account terminates too. If MiniM service was run under this user account, server terminates. If was run any processes under this user account, this processes terminates too, for example console processes. If it is need MiniM Database Server still work after user logoff, it is required to run MiniM service under the system account and use telnet access. Service account can be setup

in Windows Control Panel, applet Services. MiniM server requires interact with desktop permission to properly handle incoming telnet connections.

Other possibility to run MiniM Database Server process under the different account in console mode is running using special Windows functionality - go to Windows Explorer, right click on minim.exe with Shift key and select menu item to run process from selected user.

One of much important task can be working with network resources - with disks and printers shared resources. It is recommended to create special Windows user account to run MiniM Database Server and setup account permissions. Special Windows account for server can recommended to use in production environment.

One of much important things of Windows accounts is an access to other applications. For example, if MiniM process requires access to ODBC driver, administrator must differentiate User and System ODBC DSN. User ODBC DSNs are visible under current user account and System ODBC DSNs are visible for all Windows accounts. Or, in other cases, MiniM process can require connection to other databases using proprietary connection methods depended of client account. All this interoperation cases must be checked by server administrator to correctly setup MiniM Database Server account.

## 3.7 Routine Editor Keystrokes

Up	One line up.
Shift+Up	One line up with selection.
Ctrl+Up	Scroll up.
Down	One line down.
Shift+Down	One line down with selection.
Ctrl+Down	Scroll down.
Left	One character left.
Shift+Left	One character left with selection.
Ctrl+Left	One word left.
Shift+Ctrl+Left	One word left with selection.
Right	One character right.
Shift+Right	One character right with selection.
Ctrl+Right	One word right.
Shift+Ctrl+Right	One word right with selection.
PgDn	One page down.
Shift+PgDn	One page down with selection.

Ctrl+PgDn	Go to end of current page.
Shift+Ctrl+PgDn	Go to end of current page with selection.
PgUp	One page up.
Shift+PgUp	One page up with selection.
Ctrl+PgUp	Go to start of current page.
Shift+Ctrl+PgUp	Go to start of current page with selection.
Home	Go to line start.
Shift+Home	Go to line start with selection.
Ctrl+Home	Go to text start.
Shift+Ctrl+Home	Go to text start with selection.
End	Go to line end.
Shift+End	Go to line end with selection.
Ctrl+End	Go to end of text.
Shift+Ctrl+End	Go to end of text with selection.
Ins	Switch insert mode - insert on typing or overwrite.
Ctrl+Ins,	Copy selection into clipboard.
Ctrl+C	
Shift+Del,	Cut selection into clipboard.
Ctrl+X	
Shift+Ins,	Paste text from clipboard.
Ctrl+V	
Del	Delete one character right of caret.
BkSp,	Delete one character left of caret.
Shift+BkSp	
Ctrl+BkSp	Delete one word left of caret.
Alt+BkSp,	Undo last editing.
Ctrl+Z	
Shift+Alt+BkSp,	Redo next editing.
Shift+Ctrl+Z	
Enter,	Insert one line.
Shift+Enter,	
Ctrl+M	
Tab	Insert tabulation.
Shift+Tab	Delete left up to prior tabulation.
Ctrl+A	Select all text.
Shift+Ctrl+I	Indent selection.
Shift+Ctrl+U	Unindent selection.
Ctrl+N	Open new editor with empty text.
Ctrl+O	Open existing routine in new editor.
Ctrl+T	Delete current word.
Ctrl+Y	Delete current line.

Shift+Ctrl+Y	Delete up to end of line.
Ctrl+0,	Go to bookmark with this number.
...	
Ctrl+9	
Shift+Ctrl+0,	Create bookmark with this number.
...	
Shift+Ctrl+9	
Ctrl+U	Make selection upper case.
Ctrl+L	Make selection lower case.
Shift+Ctrl+B	Go to next pair parenthesis.
Shift+Ctrl+N	Switch selection mode to normal.
Shift+Ctrl+	Switch selection mode to column selection.
Shift+Ctrl+L	Switch selection mode to line selection.
Ctrl+F9	Compile current routine.
F9	Continue process execution.
Ctrl+F2	Halt debugged process.
F5	Set or remove debugger breakpoint at current source location.
Ctrl+Alt+B	Show list with currently defined debugger breakpoints.
Ctrl+Alt+S	Show window with process stack state.
Ctrl+Alt+V	Show list of debugged process system variables.
Ctrl+Alt+W	Show watches list.
Ctrl+F5	Add new watch.
F8	Continue process execution by one line on current stack level.
F7	Continue process execution by one line including nested stack levels.
F4	Continue process execution until current caret location or other breakpoint reached.
F6	Switch to pair routine (INT - <i>i</i> MAC or MAC - <i>i</i> INT)

### 3.8 MiniM Server Connect

MiniM Database Server provide special module MiniM Server Connect to connect to server and exchange data implemented as minimsc.dll. This library can be placed in current client application subdirectory or in any subdirectory from the PATH environment variable.

In general MiniM Server Connect interface is declared using C language.

Examples how to use minimsc.dll from other languages are optional and secondary. In real applications developer must decide what coding rules and data transfer methods must be used. If developer use UNICODE Delphi developer must correct installed file minimscint.pas, because this file was developed for ANSI Delphi.

Also developer must use characters encoding conversions to correlate client side and server side encodings and sorting.

MiniM Server Connect module can be used by client application using static and dynamic linking. MiniM installs examples to connect to MiniM from C++ and ObjectPascal (Delphi environment) using static linking. What method must be used - static or dynamic an application developer must choose too.

Library minimsc.dll is intended to create connection object to connect to MiniM process and interchange data using internal protocol over tcp/ip. On the client side this protocol handles minimsc.dll and on the server side special routine ^%srv. To activate connection listener it must be started on the server by command

```
d ^%srv
```

In most cases this command can be inserted by default into server autostart script autostart.m. This protocol does not encode data been transferred. Protocol does not contain conventions about user logins and passwords. And is supposed that application developer support special application conventions about logins and passwords and application-specific connection options.

Application interface for MiniM Server Connect is declared in the special header file minimsc.h for C++ language, in special interface unit minimscint.pas for ObjectPascal, in minimsc.vb for VB.NET, in minisc.cs for C# language, for MiniM and for Cache for MUMPS language and in minimscj.java + minimscj.dll for Java language implemented as JNI/Win32. For ActivePerl environment interface is declared inside example file.

MiniM Server connect interface consist of several functions and argument passing conventions which are much close to standard Windows calling conventions. And, in most cases, if other development tool can interact with Windows environment and build application, this tool can use minimsc.dll too. For development environment with ActiveX support MiniM provide MiniM Server Connect with ActiveX interface too.



To identify one connection established this library creates internal connection object identified by special handle *HMNMConnect*. This object is created by *MNMCreateConnect* function and removes by *MNMDestroyConnect* function. Between this functions connection object still in created but not connected state. To connect object to server developer must use *MNMConnectOpen* function and *MNMConnectClose* function to disconnect from the MiniM server. Between this functions connection object still in connected state and application can exchange data, execute commands and evaluate expressions. Library support creation of several connections and connections can have the same or different connect options.

All functions to use connection returns an signed integer as an error code. If function completes successfully, function return value 1, otherwise return value 0. If function return value 0, developer can get error occurred using *MNMGetLastError* function.

Values are passed to and from connection as special united structure with two fields - data bytes and length of bytes been used. Data bytes sequence can contain any characters including zeroes and nonprintable characters. To accept data from library functions initialization of structure is unneed. Client application can use any functions for ASCIIZ strings but terminating zero bytes does not used by connection, and connection use only bytes been specified in counter and return values are not zero terminated.

In general, main operations with connection are read expression evaluated, execute line of commands, write variable or kill variable.

Function

```
int MNMSCPROC MNMRead( HMNMConnect pConnect,
    MINIMSTR* Expression,
    MINIMSTR* Result);
```

Data bytes of *Expression* argument must contain expression in MUMPS language to evaluate on the server side. And argument *Result* accepts evaluated value.

Function

```
int MNMSCPROC MNMWrite( HMNMConnect pConnect,
    MINIMSTR* VarName,
    MINIMSTR* VarValue);
```

Data bytes of *VarName* argument must contain name of variable to assign value of *VarValue* argument.

Function

```
int MNMSCPROC MNMExecute( HMNMConnect pConnect,
    MINIMSTR* Commands);
```

Function executes line of commands in MUMPS language specified in the *Commands* argument.

Function

```
int MNMSCPROC MNMKill( HMNMConnect pConnect,
    MINIMSTR* VarName);
```

This function kill variable specified in *VarName* with subscripts. Really server executes kill with argument indirection and variable *VarName* can contain several names comma delimited.

Function

```
int MNMSCPROC MNMExecuteOutput( HMNMConnect pConnect,
    MINIMSTR* Commands);
```

This function executes line of commands in MUMPS language specified in the *Commands* variable. If this execution directly or indirectly writes to current device, all bytes been written are passed to callback function was specified by *MNMSetOutput* function. Part of bytes to transfer does not defined by size or by delimiter and passed by portions as is. This notify handler can be fired for all data, for several strings or for each byte and length of portions does not defined. In this mode write commands cannot write to current device only one byte, with zero code (\$c(0)) because it is command execution terminator. In general, this mode is used to intercept terminal-oriented output, for example result of routine compilation.

Functions

```
int MNMSCPROC MNMListGet( MINIMSTR* List,
    int pos, MINIMSTR* Element);
```

```
int MNMSCPROC MNMListSet( MINIMSTR* List,
    int pos, MINIMSTR* Element);
```

```
int MNMSCPROC MNMListLength( MINIMSTR* List);
```

are intended for list structures handling. This functions does not requires MiniM connect is established and been created. Function *MNMListGet* return into *Element* variable one list element from position *pos* from list *List*. Function *MNMListSet* replaces or adds to list *List* as element in position *pos* value of *Element*. And function *MNMListLength* returns count of list elements are available in the list *List*.

Function

```
int MNMSCPROC MNMText( MINIMSTR* Source,
  MINIMSTR* Target);
```

replaces within *Source* double quote characters and nonprintable characters to be a correct MUMPS syntax string. Result is returned into *Target* variable. If encoding executes successfully, function returns value as nonzero and if result cannot be placed into *Target*, result is zero.

Examples how be encoded byte sequences given by C notation:

```
"abcd"      ->  "abcd"
"ab\"cd"    ->  "ab\"cd"
"ab\r\ncd"  ->  "ab"_$C(13,10)_"cd"
```

Demo examples how MiniM Server Connect are installed into the *minimsc* subdirectory with name derived from used language or development tool.

To install file *minimsc.dll* with real client application any other files are not required and any registration or registry records are not required.

MiniM Server Connect can be used in several modes to get data from MiniM server-side process.

### Read expression value mode

Client side gets data as result of one function *MNMRead* call. As an argument can be used any evaluatable MUMPS expression including simple constants, one variable name, function calls and compound expression composed using operators. This expression must be constructed using a valid MUMPS syntax. Function passes text of expression to server and read back result. Result can contain any characters including zeroes and nonprintable characters.

### Intercept output mode

Client side turn this mode on if call *MNMExecuteOutput* function. After return from this function data transfer mode is turned off to simple read

expression mode. Function passes to server string of commands to execute. If before last of this commands executes been any write to current device, this data will be sent to client and fired *Output* event handler setted up by function *MNMSetOutput*.

Connection object fire event handler as data from server been received and does not distinguish any terminators and does not use any length to make parts of data. Fired event handler can accept several *write* commands arguments as such as be fired several times to one *write* command. Client side of connection use this data as a text data and data cannot contain a zero byte ( $\$c(0)$ ). Server side send data without waiting any confirmation from client side. Server can send unlimited bytes count.

In general, this mode is used to intercept output of terminal-oriented utilities. All output been written is transferred to a client side and passed to *Output* event handler.

Data transfer speed for this mode depends on network speed because server side does not wait client return.

### Group read mode

Group read mode is combination of group read event handler setted up by *MNMSetGroupRead* function and of execution on the server special subroutine to pass data to client:

```
d wo~%srv(data)
```

This subroutine pack data and send to a client side. And each of this portion fire event handler of group read mode. Handler is fired for entire data portion been sent and data can contain any bytes. Server side does not wait confirmation from the client and continues work after data been sent.

In general, main target for this data transfer mode is a packet data send for set of data strings for tables, grids or lists. This mode can be used if one function must return not single value, but set of values.

Data transfer speed for this mode depends on network speed because server side does not wait client return.

### Callback mode

Callback mode is a combination of callback read event handler setted up by *MNMSetCallback* function and of execution on the server side special function to pass data to client and get back the answer from client:

```
$$cb~%srv(command)
```

Here data passed by the *command* can contain any bytes including zeroes and nonprintable characters and passed to callback event as entire portion. If this data been received by the client side of connection, the callback event is fired and connection wait result to pass answer back to the server side. Server side will wait while client side create the answer.

Both sides, client and server, supports compliance of stacks to execute callback event handlers and connection allow recursion for callback calls. After server side accepts appropriate answer, it received answer from client as return of function been called:

```
$$cb~%srv(command)
```

In general, this data transfer mode is used for user query, or control part of client program and if this controlling can call server side back recursively.

Data transfer speed for this mode not much depends on network speed because main time is used by waiting response.

Group read and callback mode are transfered to client data entirely, and can include any bytes. Callback mode answer can include any bytes too. What data need to be sent and bytes encoding application programmer can choose in depends of application developed, for example, strings with delimiters or lists in *\$listbuild()* function encoding.

### 3.9 MiniM Server Connect, ActiveX

ActiveX component library *minimscx.dll* is an ActiveX interface to MiniM Server Connect library and converts all function and data transfer modes to an OLE automation interface. To work on the client computer it is required both files *minimsc.dll* and *minimscx.dll* must be installed and file *minimscx.dll* must be registered after installation as

```
regsvr32 minimscx.dll
```

In depends of installation mode there can be used several switches of *regsvr32* utility, or can be used any other installer who can register ActiveX library.

MiniMSCX library implements two ActiveX objects:

```
MiniM.ServerConnect
MiniM.ServerString
```

The *MiniM.ServerConnect* object is intended to implement connection with MiniM Database Server and *MiniM.ServerString* is intended to implement data transfer with OLE automation interface. Some part of *MiniM.ServerConnect* functions have doubled functions with *Str* suffix. If main function use *MiniM.ServerString* objects to pass data, simplified functions use simple strings to pass data. The *MiniM.ServerString* object can contain any bytes which cannot be represented in simple strings, for example, lists encoded in *\$lb()* function format.

### MiniM.ServerConnect functions

**CreateConnect( server, port, database)**

<i>server</i>	String, specify computer name where work MiniM Database Server
<i>port</i>	Integer, specify port, which is listen by server-side connections, in general it is 5000
<i>database</i>	String, specify the database name to make current

Function return value 1 if connection have been crated successfully, otherwise generate exception. After connection creation this connection still inactive and not in connected state. To connect it is required call *Open* function. After end of using connection object must be destroyed by *DestroyConnect* function.

### Open

Function have no any arguments and return 1 if connection connected successfully or 0 if connect failed. If before this function real connection have not been created, function generate an exception about connection object does not exists.

### Close

Function have no any arguments and returns value 1 if connection de-connected successfully or 0 if disconnect failed. If before this function real connection have not been created, function generate an exception about connection object does not exists.

### DestroyConnect

Function have no any arguments and return values. If before this function connection object have been created, this connection object disconnects if need and is destroyed.

#### GetLastError

Function have not any arguments and returns one string with last error occurred in connection.

#### Read( Expression, Value )

Function reads from the server value of expression specified as *Expression* and returns value into *Value* argument. Return value is 1 if function executes successfully, otherwise return 0.

Expression	Object of MiniM.ServerString, specify MUMPS expression to evaluate
Value	Object of MiniM.ServerString, accepts value of expression after evaluation

Return value *Value* can contain any bytes including zeroes and nonprintable characters, particularly values of lists in *\$lb()* function format.

#### ReadStr( Expression)

Function is pair to *Read* function, have one string argument with MUMPS expression to evaluate and returns string with value.

Expression	String with MUMPS expression to evaluate
------------	--

Return value cannot contain zero bytes.

#### Write( VarName, VarValue)

Function assigns to variable specified in *VarName* value specified in *VarValue*. Function returns value 1 if executes successfully and otherwise 0.

VarName	Object of MiniM.ServerString, specify variable name in MUMPS syntax to assign
VarValue	Object of MiniM.ServerString, specify value to assign to variable

`WriteStr( VarName, VarValue )`

Function is pair to function *Write* and accepts arguments as simple strings. Function return 1 if executes successfully and otherwise return 0.

`Execute( Commands )`

Function executes specified in *Commands* argument line of MUMPS commands. Argument *Commands* must be an object of *MiniM.ServerString*. Function return 1 if executes successfully and otherwise return 0.

`ExecuteStr( Commands )`

Function is pair to function *Execute* and accepts argument as a simple string. Function return 1 if executes successfully and otherwise return 0.

`ExecuteOutput( Commands )`

Function executes line of MUMPS commands specified in the *Commands* argument. The *Commands* argument must be an object of *MiniM.ServerString*. Function returns value 1 if executes successfully and otherwise returns 0. All data server process writes to current device is transferred to client and fired as parts with undetermined length on *OnTerminalOutput* event. This function differs from *Execute* by data writes interception. If function return control, the data transfer mode is switched back into normal without interception.

`ExecuteOutputStr( Commands )`

Function is a pair to *ExecuteOutput* function and accepts argument as a simple string.

`Kill( VarName )`

Function accepts an object of *MiniM.ServerString* with variable name specification to kill this variable with subscripts. Subscripts can be defined as evaluatable expressions using MUMPS syntax. This function execute *kill* command with indirection and argument can contain several variable names to kill and be a comma delimited. Function return 1 if executes successfully and otherwise return 0.



KillStr( VarName )

Function is a pair to *Kill* function and accepts argument as a simple string. Function return 1 if executes successfully and otherwise return 0.

ListLength( List )

List            Object of MiniM.ServerString, specify value of list, encoded as *\$lb()* function format.

Function return integer with number of elements in the *List* and function counts defined and undefined list elements.

ListGet( List, Pos, Elem)

Function return value of list *List* element from position *Pos* into variable *Elem*.

List            Object of MiniM.ServerString, threats as a list encoded as *\$lb()* function format.  
 Pos            Integer, specify element position  
 Elem           Object of MiniM.ServerString, accepts list element's value

Function return 1 if executes successfully and otherwise return 0.

ListSet( List, Pos, Elem)

Function creates or replaces list element of *List* in position *Pos* with value of *Elem*.

List            Object of MiniM.ServerString, threats as a list encoded as *\$lb()* function format  
 Pos            Integer, specify element position  
 Elem           Object of MiniM.ServerString, value to place into list

Function return 1 if executes successfully and otherwise return 0.

Text( Value, Result )

Function decorate string specified in the *Value* argument to be a valid string representation in MUMPS syntax.

Value           Object of MiniM.ServerString, value to encode  
 Result          Object of MiniM.ServerString, result of encoding

Function return 1 if executes successfully and otherwise return 0.

`TextStr( Value)`

Function is a pair to *Text* function and have simplified interface. Function accepts and returns simple strings.

#### Events of object `MiniM.ServerConnect`

`OnTerminalOutput( Value )`

**Value** String, part of data been written to current device while functions *ExecuteOutput* or *ExecuteOutputStr* executes

This event is fired if client side accepts data from server written to current device while functions *ExecuteOutput* or *ExecuteOutputStr* executes. Connection object does not change bytes received and does not split by terminators or length. String written cannot contain zero byte ( $\$c(0)$ ) because this byte is used as an interception mode terminator. After return from *ExecuteOutput* or *ExecuteOutputStr* functions connection object restores data transfer mode into normal mode without interception.

`OnGroupRead( Value )`

**Value** Object of `MiniM.ServerString`, entire part of data, been sent from the server.

An event *OnGroupRead* is fired on the client on each time server calls

`d wo~%srv(value)`

And here entire value of *value* fully collected as byte sequence on the client and this value is passed to event *OnGroupRead*. This event is fired asynchronously and server side does not wait answer from the client. Server can pass series of data and event will be fired on each passing. Client side is not synchronized with the server and can handle event when the server side always pass set of strings and return control. In this event handler context client side cannot call server again and must return control.

```
OnCallback( Command, Answer )
```

Command	Object of MiniM.ServerString, data sent from the server to client.
Answer	Object of MiniM.ServerString, answer to return from client to server.

An event *OnCallback* is fired on the client on each call on the server

```
$$cb^%srv(command)
```

And here entire value of *command* fully collected as byte sequence on the client and this value is passed as a *Command* argument of event handler. When this event is fired, server side stops execution and wait answer or other command to execute or expression to evaluate. When event handler create result and return control, connection object return answer been sent to a server side and server process continues to word and gets answer as return of function *\$\$cb^%srv*.

Whithin *OnCallback* event handler client application can call server again recursively to evaluate expressions or execute commands and this commands can call client again. Connection save bot stack contexts for calls server from client and client from server.

An objects of *MiniM.ServerConnect* does not limit *Command* and *Answer* values by format, byte contains or encoding. What does it mean an application programmer must choose in dependent of application developed.

### Functions and properties of MiniM.ServerString

#### Value

*Value* is a property of *MiniM.ServerString* object with type of simple string. This property is accessible for read and write.

#### Length

*Length* is a property of *MiniM.ServerString* object with type of Integer and contains real bytes count been used in object. This property is accessible for read and write. When assigned to value less than 0 property sets to value 0, when assigned to value greater than 32767, property sets to value 32767.

#### GetAt( Pos)

Function returns integer as character code at position *Pos*, where position is specified by integer. If *Pos* show outside of bytes are available, function returns code -1. Character code is returned as integer from 0 to 255.

**SetAt( Pos, Code)**

Function replaces byte at position *Pos* to byte with code *Code*. Arguments *Code* and *Pos* must be an integers. If value of *Pos* shows outside of bytes are available, function does nothing.

**Add( Val )**

Function concatenates to internal object data data of other object *Val* of type *MiniM.ServerString*. Value of *Val* does not changes. If function executes successfully, function return value 1, otherwise value 0.

**AddStr( Val)**

Function concatenates to internal object data bytes of string *Val*. Value of *Val* does not changes. If function executes successfully, function return value 1, otherwise value 0.

MiniM Database Server installer installs examples how to use objects *MiniM.ServerConnect* and *MiniM.ServerString* in the subdirectory VBS and examples are made using Visual Basic Script. To execute this examples run in the console of cmd the command:

```
cscript.exe //Nologo example1.vbs
```

All examples are console examples and outputs data to the screen.

## 3.10 Import / export API

MiniM Database Server implement routines for export and import globals, routines and bytecode to and from file. This section describes API for program export and import.

### 3.10.1 Global import

#### Function

```
$$import~%GI(fname,fmt,.list,show)
```

#### Arguments

fname	File name to import globals data from
fmt	Globals export format number
list	Local variable name (by reference) to accept imported globals names as subscripts (optional)
show	Write (1) or not (0) to current device import report (optional, by default 0)

### Global formats

Argument *fmt* can be one of the following:

- 1 Stream format (Cache or MSM), line feed symbols are reserved by format.
- 2 Variable length records format (Cache or MSM), data and subscripts can contain any characters.

### Return value

Function return count of imported globals with different names. If function fails, return value is 0.

### Example

```
s fname="e:\Data.gsa"
s fmt=2
s show=1
s count=$$import~%GI(fname,fmt,.list,show)
```

## 3.10.2 Block global import

### Function

```
$$import~%GBI(filename)
```

### Arguments

filename File name to import globals from.

### Return value

If function completes successfully, return value is 0. Otherwise function return internal error code with following comma and error description. Ex-

ample:

```
1,Failed to read or write file specified
2,Invalid MiniM block export file
```

On this import function does not write anything to current device.

### 3.10.3 Routine import

#### Function

```
$$import^%RI(fname,.rlist,compile,show)
```

#### Arguments

fname	File name to import routines from
rlist	Local variable name (by reference) to write imported routine names to subscripts (optional)
compile	Indicator need (1) or not (0) compile imported routines after import (optional, by default 1)
show	Indicator need (1) or not (0) report import stages to current device (optional, by default 0)

#### Return value

Function return number of routines imported.

#### Example

```
s fname="e:\matr.rou"
s compile=1
s show=0
s count=$$import^%RI(fname,.rlist,compile,show)
```

### 3.10.4 Bytecode import

#### Function

```
$$import^%RIMF(fname,.list,show)
```

#### Arguments



fname	File name to import bytecode from
list	Local variable name (by reference) to write imported bytecode names as subscripts (optional)
show	Indicator report (1) or not (0) import stages to a current device (optional, by default 0)

**Return value**

Function return number of bytecodes have been imported.

**Example**

```
s fname="e:\query.mmo"
s show=1
s count=$((import~%RIMF(fname,.list,show))
```

**3.10.5 Globals export****Function**

```
$$export~%G0(source,fname,descr,fmt,show)
```

**Arguments**

source	Variable name (local or global) which subscripts contains globals names to export
fname	File name to export globals to
descr	Export description (stored in the file)
fmt	Export format number
show	Report (1) or not (0) export stages to current device (optional, by default 0)

**Export formats**

1	Format Cache, stream, line feed characters are used by format
2	Format Cache, variable length records, data and subscripts can contain any characters
3	Format MSM, stream, line feed characters are used by format

- 4           Format MSM, variable length records, data and subscripts can contain any characters

If description contains only one character circumflex (^) and format is 1 or 3, export function creates special header as autoimport header. This file after export can be imported later from OS command line:

```
minim.exe < globalexport.gsa
```

### Return value

Function return number of globals exported.

### Example

```
s names("^Data")=""
s names("^Global2")=""
s fname="e:\gdata.gsa"
s descr="Daily export"
s fmt=4
s show=0
s count= $$export ^%G0($na(names),fname,descr,fmt,show)
```

## 3.10.6 Block global export

### Function

```
 $$export ^%GB0(name,fname,descr)
```

### Arguments

name	Local or global variable name which subscripts contains globals names to export.
fname	File name to export globals to.
descr	Description, this text stored in the export file.

### Return value

If function executes successfully, return value is 0. Otherwise function return nonzero value with internal error code and followed comma and error text, for example:

```
1,Failed to read or write file specified
2,Invalid MiniM block export file
```

Example how to use block global export:

```
s gnames("Data")=""
s gnames("Ind")=""
s filename="e:\data.g"
s comment="Daily export"
s err=$$export~%GB0($na(gnames),filename,comment)
```

This function does not write anything to current device.

### 3.10.7 Routine export

#### Function

```
$$export~%R0(source,fname,descr,show)
```

#### Arguments

source	Local or global variable name which subscripts contains routine names to export.
fname	File name to export routines to.
descr	Export description, stored in the file.
show	Indicator report (1) or not (0) export stages to current device (optional, by default 0).

If description contains only one character of circumflex (^), export function writes the special header for autoimport. Later this file can be imported from command line as:

```
minim.exe < routineexport.rsa
```

#### Return value

Function return number of exported routines.

#### Example

```

s rounames("query")=""
s rounames("order")=""
s show=0
s descr="Daily export"
s fname="e:\routines.rsa"
s count=$$export^%R0($na(rounames),fname,descr,show)

```

### 3.10.8 Bytecode export

#### Function

```

$$export^%ROMF(fname,.rounames,descr,show)

```

#### Arguments

fname	File name to export bytecodes to.
rounames	Local variable name (by reference) which subscripts contain bytecode names to export.
descr	Description, stored in the export file.
show	Indicator, report (1) or not (0) export stages to current device (optional, by default 0).

#### Return value

Function return number of bytecodes have been exported to file.

#### Example

```

s rounames("query")=""
s rounames("order")=""
s descr="Daily export"
s show=0
s fname="e:\routines.mmo"
s count=$$export^%ROMF(fname,.rounames,descr,show)

```

## 3.11 Routine Change API

MiniM Database Server implements special API to change routines and get special information about routines. Subroutines are placed into special system routine ^%R. Subroutines uses routine names which can contain routine

types as routine name extensions. If extension does not specified, subroutines assumes INT type. Routine names are case sensitive but routine types are not. On error subroutines return value of 0.

Next examples uses routine `first.inc` with the following content:

```
; header line
w "Hello from include",!
w 1234,!
```

### Load routine lines into variable

Prototype:

```
s err=$$LOAD^%R(rouname,$na(text))
```

Subroutine writes lines of routine *rouname* from database storage into specified local or global variable *text* with numeric subscripts. Subroutine does not writes special information. Before execution subroutine clears *text* variable.

Example:

```
USER>s err=$$LOAD^%R("first.inc",$na(rou("rou")))
USER>w
err=1
rou("rou",1)=" ; header line"
rou("rou",2)=" w "Hello from include",!"
rou("rou",3)=" w 1234,!"
```

### Save routine lines from variable

Prototype:

```
s err=$$SAVE^%R(rouname,$na(text))
```

Subroutine creates or overwrites routine with lines from subscripts of specified in *text* local or global variable. Subroutine uses lines in index order as they are follows in the *text* variable, and integer numeric subscripts are not mandatory.

Example:

```

USER>s rou("a")=" ; comment"
USER>s rou("b")=" w 1234,!"
USER>s rou("c")=" w 7896,!"
USER>s err=$$SAVE^%R("first.inc",$na(rou))
USER>w
err=1
rou("a")=" ; comment"
rou("b")=" w 1234,!"
rou("c")=" w 7896,!"

```

After execution this code creates routine `first.inc` with the following content:

```

; comment
w 1234,!
w 7896,!

```

### Change line in routine

Prototype:

```
s err=$$SETLINE^%R(rouname,lnum,line)
```

Subroutine rewrites in the routine `rouname` line with number `lnum` by value of `line`. Lines in `lnum` counts from 1.

Example:

```
USER>s err=$$SETLINE^%R("first.inc",1," ; changed comment")
```

After execution this subroutine routine `first.inc` got the following content:

```

; changed comment
w 1234,!
w 7896,!

```

### Get number of lines

Prototype:

```
s err=$$GETCOUNT^%R(rouname)
```

Subroutine returns number of lines in the routine *rouname*.

#### Get one line of routine

Prototype:

```
s err=$$GETLINE^%R(rouname,lnum,.line)
```

Subroutine return into the *line* value of line of routine *rouname* with number *lnum*. Lines in *lnum* counts from 1.

Example:

```
USER>s err=$$GETLINE^%R("first.inc",2,.line)
USER>w
err=1
line=" w 1234,!"
```

#### Insert line into routine

Prototype:

```
s err=$$INSLINE^%R(rouname,lnum,line)
```

Subroutine inserts one line into routine *rouname* from position *lnum* and with content of *line*. All prior available lines from position *lnum* shifts by one. Lines in *lnum* counts from 1.

Example:

```
USER>s err=$$INSLINE^%R("first.inc",2," #; second comment")
```

After execution this subroutine routine *first.inc* got the following content:

```
; changed comment
#; second comment
w 1234,!
w 7896,!
```

#### Removing one line from routine

Prototype:

```
s err=$$DELLINE^%R(rouname,lnum)
```

Subroutine removes in the routine *rouname* one line of code at position *lnum*. All available lines followed by this line shifts by one position. Lines in *lnum* counts from 1.

Example:

```
USER>s err=$$DELLINE^%R("first.inc",3)
```

After execution this subroutine routine *first.inc* got the following content:

```
; changed comment
#; second comment
w 7896,!
```

### Get routine modification timestamp

Prototype:

```
s hor=$$DATE^%R(rouname)
```

Subroutine returns date and time of routine *rouname* change timestamp in the *\$horolog* system variable format. In the case this routine does not exists, subroutine returns an empty string.

Example:

```
USER>w $$DATE^%R("first.inc")
62136,59140
```

### Get bytecode modification timestamp

Prototype:

```
s hor=$$BCDATE^%R(rouname)
```

Subroutine returns date and time of bytecode for routine *rouname* change timestamp in the *\$horolog* system variable format. In the case this bytecode does not exists, subroutine returns an empty string.

Example:



```
USER>w $$BCDATE^%R("first.inc")
USER>w $$BCDATE^%R("%MPP")
62131,58917
```

In the MiniM Database Server routine changes does not affect to compiled bytecode immediately and subroutines of the ^%R routine does not change bytecode. To compile routines MiniM Database Server implements special subroutine

```
d compile^%RCOMPIL(rouname)
```

Subroutine does not compile INC routines. If routine have the MAC extension, this subroutine call macro preprocessor before compilation and produces the INT routine. Next routine compiles into executable bytecode. While subroutine works, all messages with errors writes into current output device.



# Chapter 4

## CHUI Utilities

CHUI is an abbreviation of CHaracter User Interface. It is an applications for interactive alphanumeric environment, for example console or telnet. Applications use one screen to display text with formatting, with possible color decoration and with possible pseudographic symbols.

CHUI utilities can be used with local console access and with remote telnet access from any other computer connected via tcp/ip network. Utilities traditionally made in very simple interface to execute simple actions in any possible situation. CHUI utilities in most cases have not rich interface and are only action-oriented.

CHUI utilities are only simple interface to internal small applications which are runs on the server, not on the client side. And administrator or other operator must understand what disks, directories and files are used - it is resources of server only or visible from server. And, of course, server process must have enough permission to access this resources.

### 4.1 %BACKUP

Utility `^%BACKUP` provided for backup one or more or all databases of MiniM Database Server.

Usage

```
do ^%BACKUP
```

Utility offer to choose backup type - full or differential. This utility use hot backup and can be used with active processes in transaction states.

MiniM Backup utility.

Select backup type:

- 1) Full backup
- 2) Differential backup

Select option:

Utility wait operator response - 1 or 2 and if entered other alternative, utility stop working.

Next utility offer to choose databases to backup

Select databases to backup.

- 1) All databases
- 2) Selected databases

Select option:

With selection 1 (All databases) utility will backup all databases configured and have not Autocreate indicator (Autocreate = 0).

If operator choose 2 alternative (Selected databases), utility list all available databases and offer to select database numbers to backup.

List of available databases:

- 1) USER
- 2) %SYS
- 3) TEMP

Select db number to add to backup:

After selection database by number this database is placed into backup list and utility offer to select one more database.

List of available databases:

- 1) USER
- 2) %SYS
- 3) TEMP

Select db number to add to backup: 1

List of selected databases:

USER

List of available databases:

- 2) %SYS
- 3) TEMP

Select db number to add to backup:

If operator select unexistent number (or simply press Enter) utility ends database selection. If no any databases selected, utility stop working. Otherwise utility offer to enter file name to save backup.

List of selected databases:

USER

List of available databases:

2) %SYS 3) TEMP

Select db number to add to backup:

Enter file name to backup to:

If operator enter empty string, utility stop working. File name entered must be accessible from server. Utility have no limitation about file name or extension and operator can enter unexisting directories. Utility check directory and create ones if need.

Next utility offer to select journal operation to do after backup:

Journal truncate option:

1) Truncate journal

2) Keep journal as is

Select option:

On selection 1 (Truncate journal) utility removes all journal records before most earlier for current opened transactions. If operator select 2 (Keep journal as is) utility does not make any operations with journal records.

Next utility offer to select reporting option.

Report option:

1) Make report

2) No report

Select option:

If operator select 1 (Make report) utility will make detailed report of all what happens.

If operator select option *Make report*. utility ask file name to write report.

Report option:

1) Make report

2) No report

Select option: 1

Enter file name to print backup report to:

If operator enter empty file name, utility make detailed report not to file, but to current screen.

Report content is dependent of actions performed, operator selections and time to execute backup stages can differs depending of data available. For example, if operator select differential backup for one USER database, report can seems like:

```
Start backup modified databases.
Start backup modified database USER
Start finish backup phase.
Finish backup USER
Backup done successfully.
```

## 4.2 %DBCLEAN

Utility ^%DBCLEAN performs database blocks scan, searches unused space in blocks and cleans this space.

Usage

```
do ^%DBCLEAN
```

Utility offer to select database from available databases list to cleanup.

```
MiniM database cleanup utility
Utility cleans unused database space.
All used data will not be touch.
```

```
Available database list:
1) USER 2) %SYS 3) TEMP
Select database number to clean:
```

If operator choose unexisting alternative (or simply press Escape or Enter) utility stop working. Otherwise utility start to clean database.

Utility reads database blocks sequentially, but changes only some block types, which may have unused space. Each block before read is locked to write.

Time to work depends of number of pages in datafiles. Utility reads all pages into page cache and can displace pages already read by other processes

to walk globals and this processes requires reread displaced pages again. MiniM Database Server use LRU page caching algorithm and checking can devaluate pages used.

Sample utility screen:

```
TEMP>d ^%DBCLEAN
```

```
MiniM database cleanup utility
Utility cleans unused database space.
All used data will not be touch.
```

```
Available database list:
 1) USER 2) %SYS 3) TEMP
Select database number to clean: 3
Cleanup database TEMP
Database TEMP contains 128 blocks.
Cleanup utility ends.
```

Utility may be run from any database and for any database which have write access.

## 4.3 %DBCRC

Utility ^%DBCRC is designed to check datafiles pages using cyclical redundancy check.

Usage

```
do ^%DBCRC
```

Utility offer to select database from available databases list to check.

```
MiniM database CRC check utility
```

```
Available database list:
 1) USER 2) %SYS 3) TEMP
Select database number to check:
```

If operator choose unexisting alternative (or simply press Escape or Enter) utility stop working. Otherwise utility start to check datafiles pages.

Cyclical redundancy check is present in each datafile page and is created on write to datafile. Checking allow to find store errors in used filesystem or disk phisical errors, is page stored correctly or not.

Time to work depends of number of pages in datafiles. Utility reads all pages into page cache and can displace pages already read by other processes to walk globals and this processes reguires reread displaced pages again. MiniM Database Server use LRU page caching algorithm and checking can devaluate pages used.

If utility detect CRC errors, utility display information about error detected. This problem is related to all database and administrator must check the reason disk hardware failure and restore databases from backup.

## 4.4 %DBSIZE

Utility ^%DBSIZE display current database file in megabytes and grows limit and can expand current database volume.

Usage

```
do ^%DBSIZE
```

Utility offer to select option to perform - show current database size or add datafiles space.

```
USER>d ^%DBSIZE
```

```
MiniM Show database size utility
```

```
Select utility option:
```

- 1) Show current database size and limit
- 2) Extend current database size

```
Select option:
```

If operator choose alternative 1 utility show current database sizes and currently defined grows limits. All values are shown in megabytes.

Before show list utility offer to enter optional file name to store report. If name is not an empty string, utility store report to file specified.

After execution utility offer to enter option again. Sample of utility screen:



MiniM Show database size utility

Select utility option:

- 1) Show current database size and limit
- 2) Extend current database size

Select option: 1

Enter file name to print to (optional):

Database	Size (MB)	Limit (MB)
USER	11	unlimited
%SYS	1	1024
TEMP	6	1024

Select utility option:

- 1) Show current database size and limit
- 2) Extend current database size

Select option:

If operator select option 2, utility show list of available databases and offer to select database to expand.

Next utility offer to enter number of megabytes need to add to database. After selection utility expand database to number of megabytes have been chosen. Utility expand databases independently of current autoexpand database option. Sample of utility screen:

Select utility option:

- 1) Show current database size and limit
- 2) Extend current database size

Select option: 2

Available databases:

1) USER 2) %SYS 3) TEMP

Select database number to extend: 3

Enter megabytes count to add to database TEMP : 2

Database extension done.

Utility ^%DBSIZE is intended to control database size used on disk and for manual database expanding. Utility can expand databases without auto-

expand option. Utility does not expand databases over configured database grows limit and can expand only to this limit.

## 4.5 %GBI

Utility ^%GBI is developed to import globals data from block global import file format or show this file information.

Usage

do ^%GBI

Utility on start offer to enter file name and select action to do:

```
USER>d ^%GBI
MiniM block global import
Enter file name to import globals from: e:\blocks.g
Select operation:
  1. Show export comment
  2. Show global's names
  3. Import globals
Option:
```

If operator choose 1, utility show file comment, entered on export to this file.

If operator choose 2, utility show exported globals name, which was exported to this file.

If operator choose 3, utility imports globals data from file specified.

Utility can import data only from file created by MiniM block global export only, no other MUMPS systems.

Utility imports not pages been exported, but globals data only. Utility imports data into current database using current process environment, with current journaling mode. So, operator can import from block global export many times step-by-step.

Utility imports data as the *merge* command, global names with same subscripts values are replaced, and new globals inserted. If database contains globals names which are absent in block global export file, this names does not removed.

Block global import can work slowly than block global export, because import parses page been exported and execute all operations to insert globals data including journaling. Block import time and block export time also can differs dependent of current page cache memory used.

Block global import executes not atomicity to all file data, utility imports data as the *merge* command, without blocking all other processes.

Utility uses current subscript collation table defined to walk globals subscripts.

## 4.6 %GBO

Utility ^%GBO executes block globals export to file for selected globals.

Usage

do ^%GBO

Utility offer to enter file name to export globals data to. If this file already exists, this file will be overwritten. After file name utility offer to enter file export comment. This is optional string to add to file and can contain any information about this export.

```
USER>d ^%GBO
MiniM block global export
Enter file name to export globals: e:\databl.k.g
Enter description:
```

Next utility offer to enter globals name mask to export. To end global mask entering operator must enter empty string (simply press Enter).

```
Global name mask: GloData
Global name mask:
```

After input global names mask utility execute block global export for selected globals.

Block global export differs from other globals formats, here export only significant page areas with keys and data for later block global import. Export use internal undocumented format, with the same format used to store globals in datafile pages. All internal block export and import functions

`$v("db")` are not documented too and can be changed in later MiniM versions. Block global export executes only from current database.

Block export executes at one moment point of view, in the same state on export start. While block global export executes, all other processes cannot change this global and waits export done. Atomic export executes only for one global. If utility must export several globals, next global will be exported in state at the end of prior global exported.

Block global import is intended only for MiniM usage and does not accepted by other MUMPS implementations. Other MUMPS systems can use his own block export files format. So, administrator can transfer block exported data only between two or more MiniM Database Servers.

Block global export executes much faster than other global export because system does not use any data or subscripts parsing and writes leaf pages entirely and use only significant page space.

## 4.7 %GDIR

Utility `^%GDIR` is developed to list globals are available in the current database.

Usage

```
do ^%GDIR
```

Utility offer to enter globals name mask. Name mask can contain symbol "?" as pattern of one any character and symbol "\*" as pattern of any characters sequence.

Utility sequentially ask several name masks. If operator enter empty string (press Enter or Escape), utility stops to enter name masks.

Next utility query optional file name to output list of globals names are available. If operator enter empty string (press Enter or Escape), utility show globals names on the screen, otherwise outputs this list into file specified.

Example:

```
Global name mask: *
Global name mask:
Enter file name to print to (optional):
List of selected globals's in database USER
```

```

^A
^D
^ROUTINE
^UTILITY
^a
^rOBJ

```

```
USER>d ^%GDIR
```

```
MiniM Globals listing utility
```

```
Global name mask: *0*
```

```
Global name mask:
```

```
Enter file name to print to (optional):
```

```
List of selected globals's in database USER
```

```
^ROUTINE
```

```
^rOBJ
```

## 4.8 %GI

Utility ^%GI perform global import into current database global data from global export file.

Usage

```
do ^%GI
```

Utility offer to enter file name to import data from. If operator enter an empty string (simply press Enter or Escape), utility exits.

Next utility offer to enter global export format number, which have been used for export to this file.

Select file format:

1. Stream format (Cache or MSM or ANSI)
2. Variable-length format (Cache or MSM)

Format option:

Utility use different unpacking methods depended of selected format code. IF operator choose unsupported format number, utility exits.

Utility uses already used for different MUMPS systems global export formats, intended by Intersystems (Cache) and Micronetics (MSM) and widely used by many MUMPS servers.

Stream formats can be used to store data which have not in data or in subscripts any unprintable characters. If at least one subscript or global data can contain any characters, operator must export this globals using variable-length formats. Stream formats uses line feed character to distinguish parts of names and data, and variable-length formats contain special market to indicate data length have been used.

Utility imports data without removing any globals with the same names and override data if database contain the same subscripts.

Time to work is depended of global export file volume and from available page cache. Utility does not lock or block imported global names and other processes still work and can access globals for import.

## 4.9 %GL

Utility ^%GL perform global import into current database global data from global export file.

Usage

```
do ^%GL
```

Utility ^%GL is a wrapper to ^%GI utility for compatability with widely used names in other MUMPS systems.

## 4.10 %GO

Utility ^%GO perform export global data to file.

Usage

```
do ^%GO
```

Utility offer to enter file name to export globals data to. If operator enter an empty string (simply pressing Enter or Escape), utility exits.

Next utility offer to select global export format.

Select one of the following formats:

1. Cache stream
2. Cache variable length
3. MSM stream
4. MSM variable length
- Q. Quit export

Choose format:

Utility use different packing methods depended of selected format code. IF operator choose unsupported format number, utility exits.

Utility uses already used for different MUMPS systems global export formats, intended by Intersystems (Cache) and Micronetics (MSM) and widely used by many MUMPS servers.

Stream formats can be used to store data which have not in data or in subscripts any unprintable characters. If at least one subscript or global data can contain any characters, operator must export this globals using variable-length formats. Stream formats uses line feed character to distinguish parts of names and data, and variable-length formats contain special market to indicate data length have been used.

Next utility offer to enter export description or autoloader indicator to make autoloader format. Next time operator can see file header to determine what data have been exported. If operator select autoloader indicator ("^"), utility create special file header which can be used later to import globals data from file using command line:

```
minim.exe < filename.gsa
```

Moreover, if operator select autoloader option, this file can be imported as others files without autoloader header.

To make autoloader header for globals export file utility uses first two file lines intended by MUMPS standard to insert special MUMPS commands and autoloader option cannot be used with variable-length file formats.

Next utility offer to enter globals names to export.

Global name mask: A

Global name mask:

Here global indicator circumflex "^^" is optional.

If operator enter an empty string, utility stops to enter names and executes global export with selected options if this globals are present in the database.

Operator can use names masks, where symbol "?" is a pattern for one character and symbol "\*" is a pattern for any characters sequence.

Time to work is depended of globals total volume to export. Utility does not lock or block globals and other processes can work with this globals in the same time.

## 4.11 %GS

Utility ^%GS perform export global data to file.

Usage

```
do ^%GS
```

Utility ^%GS is a wrapper to ^%GO utility for compatability with widely used names in other MUMPS systems.

## 4.12 %JOBTAB

Utility ^%JOBTAB show current process list on the server, some process characteristics and allow to kill selected process.

Usage

```
do ^%JOBTAB
```

Utility show table of currently active processes on the server with columns of: 1) job's number, 2) job's current database, 3) job's currently executed routine, 4) job's current device and 5) job's last global reference.

Sample utility screen:

Job table at 02.06.2009 13:03:39

Entry	Job	Database	Routine	Principal	Last global reference
-------	-----	----------	---------	-----------	-----------------------



```

1) 812  %SYS      %srv      |TCP|      ^ROUTINE("%JOBTAB",0)
2) 908  USER     %srv      |NULL|    ^%SRV("stop")
3) 940  USER     %JOBTAB   |CON|     ^UTILITY(940,2)

```

Page #1 of 1

[R]eload | [Q]uit | kill [J]ob | kill [E]ntry | Up/Down

Command:

Utility show table of jobs with pages and allow page change, if number of currently active processes does not allow to show all table at one screen, and show current page of job's list and total number of pages.

Utility waits operator command case insensitive.

```

R          Reload list and show from first page
Q          Exit utility
J          Delete job specified by job number
E          Delete job specified by position in list shown

```

On press up and down arrows utility goes to prior or next page and refreshes the screen.

Utility obtain job's information not as a snapshot, and sequentially reads job's characteristics one-by-one. So, last line of the job's table can be creates with small of time difference. Full job's list creation is dependent of total number of jobs on the server.

## 4.13 %JOURNAL

Utility ^%JOURNAL show current journal state and control by journal switch and truncation.

Usage

```
do ^%JOURNAL
```

Utility at start show current journal state - directory and last used journal file.

Next utility offer to choose journaling option - switch current journal file or truncate journal.

Sample utility screen:

Minim Journal state utility

```
Journal directory:
e:\workfiles\minim\journal\
Current journal file:
e:\workfiles\minim\journal\2009-06-02.002
```

```
Select journal operation:
  1) Switch current journal file
  2) Truncate journal
Select option:
```

If option 1 chosen, utility switches server to the new journal file, and link to full journal sequence. If option 2 chosen, utility truncates journal, and journal file store only records need to complete (commit or rollback) currently uncompleted transactions of server processes. Other journal records are removed and can be removed prior unneed journal files.

If operator choose other option, utility ends working.

## 4.14 %LOCKTAB

Utility ^%LOCKTAB show locks are available on the server as table and allow to remove lock specified.

Usage

```
do ^%LOCKTAB
```

Utility show list of locks available on the server as table and organize table by pages. Each lock is shown with job number of owner, locking counter and name was locked. Names of locks are names of local and global variables. Sample utility screen:

```
Lock table at 02.06.2009 18:51:52
8388608 bytes total, 160 bytes used
```

```
Entry Owner    Count  Item
  1) 940         1    a
  2) 940         1    ^|"USER"|b
Page #1 of 1
```

```
[R]eload | [Q]uit | [D]elete one | delete [J]ob's >>
>> | delete [A]ll | Up/Down
```

Command:

Utility show current locking memory usage, number of table's page and total number of pages available. Next utility waits operator command case insensitive. Table of utility commands:

R	Reload list of locks and show table from first page
Q	Exits utility
D	Remove one lock specified by position in table shown
J	Remove all locks of job specified by job number
A	Remove all server locks

Operator can change page to prior or next page of the table by pressing up and down arrows.

Utility gets list of locks not as snapshot and reads list sequentially, one-by-one. So, some part of table can be incoherent.

Utility can remove locks owned by any process, not only by current process of utility.

## 4.15 %PERFMON

Utility ^%PERFMON show current server performance counters.

Usage

```
do ^%PERFMON
```

Utility clears screen and display current server performance counters available. Screen refreshes every 1 second. Performance counters are shown in table with first column as counter name, second column as current counter value and thord column as middle counter value by last 12 seconds.

Utility waits any input. If no any characters entered, utility continues display performance counters with values of current second. If operator press any key, utility simply ends working.

Sample utility screen:

MiniM Performance Monitor		
Performance Counter	Current	Middle
Blocks read/sec	0	0
Blocks write/sec	0	0
Blocks allocated/sec	0	0
Blocks freed/sec	0	0
Datafile expand MB/sec	0	0
Globals read/sec	1	1
Globals write/sec	0	0
Globals kill/sec	0	0
Locals read/sec	1206	1209
Locals write/sec	659	662
Locals kill/sec	0	0
Block split/sec	0	0
Journal write Bytes/sec	0	0
Routine read/sec	0	0

Press any key to exit

## 4.16 %RCHANGE

Utility ^%RCHANGE search string specified in routines source code and replace found with optional recompilation.

Usage

do ^%RCHANGE

Utility offer to enter pairs search string and replace string with option case sensitive search or not. Operator can enter several pairs to replace. Sample utility screen:

MiniM Replace text in routines utility

1. Search for: abc  
    Replace with: def  
    Case sensitive search? [Y/N]: n
2. Search for:

If operator enter an empty string, utility ends to input pair search - replace strings and offer to enter routines names masks to search in.

Routine name mask: \*a\*

Routine name mask:

If operator enter an empty string (simply press Enter or Escape), utility ends masks input. Operator can use special symbols - "?" as a pattern for any character and "\*" as a pattern of any characters sequence.

Next utility offer to enter recompilation option for routines to be changed after replacing

Recompile changed routines? [Y/N]:

Nect utility offer to enter file name for report.

Enter file name to print to (optional):

If operator enter an empty string, utility display report to the screnn, otherwise to file specified.

Utility show routine names where search text have been found, strings after replacing and recompilation result with possible syntax errors. If routine does not changed, this routine does not recompiled.

If routine names are entered without extension, utility use only standard routines (INT), otherwise utility use specified routine types. In the routine extension cannot be used placeholders and utility support the following types: INT, MAC, INC. Example:

Enter name masks or #L to list selected or #D to remove selection.

Routine name mask: MP\*.MAC

Routine name mask: f\*.INC

Routine name mask: #L

List of selected routine names:

MPPSRC.MAC

first.INC

-----

## 4.17 %RCOMPIL

Utility ^%RCOMPIL compiles routines selected.

Usage

do ^%RCOMPIL

Utility offer to enter routines names mask for compilation:

MiniM Compile routines utility

Enter name masks or #L to list selected or #D to remove selection.

Routine name mask: ?a\*

Routine name mask:

Utility accept list of routines masks. Operator can use special symbols - "?" as a pattern for any character and "\*" as a pattern of any characters sequence. To stop enter routine masks simply press Enter or Escape.

Next utility offer to enter file name (optional) for report. If operator enter an empty string, utility outputs report to the screen.

While utility recompile routines, report made contains current database name, routine names to recompile and syntax error descriptions. Sample utility screen:

Compile selected routines in USER

Compile label

Compile matrix

After compilation utility ends.

If routine names are entered without extension, utility use only standard routines (INT), otherwise utility use specified routine types. In the routine extension cannot be used placeholders and utility support the following types: INT, MAC, INC. Example:

Enter name masks or #L to list selected or #D to remove selection.

Routine name mask: MP\*.MAC

Routine name mask: f\*.INC

Routine name mask: #L

List of selected routine names:

MPPSRC.MAC

first.INC

-----

## 4.18 %RCOPY

Utility ^%RCOPY copies selected routines from current database into another.

Usage

do ^%RCOPY

Utility offer to enter routines names mask for copy and recompilation:

```
USER>d ^%RCOPY
```

```
MiniM Routine Copy Utility
```

```
Enter name masks or #L to list selected or #D to remove selection.
```

```
Routine name mask: CLI*
```

```
Routine name mask:
```

Utility accept list of routines masks. Operator can use special symbols - "?" as a pattern for any character and "\*" as a pattern of any characters sequence. To stop enter routine masks simply press Enter or Escape.

Next utility ask need or not recompile selected routines after copying.

```
Recompile routines after copy? [Y/N]: y
```

Next utility display the list of all available databases, current database name and query target database name to copy routines.

```
List of available databases:
```

```
%SYS
```

```
TEMP
```

```
USER
```

```
APPDB
```

```
Current database: USER
```

```
Enter target database name: appdb
```

The database name can be entered case insensitive. If operator enter an empty line, utility ends. If operator enter name of the current database, utility query target database name again.

After all selected routines was copied into target database, utility recompile this routines if need.

If routine names are entered without extension, utility use only standard routines (INT), otherwise utility use specified routine types. In the routine extension cannot be used placeholders and utility support the following types: INT, MAC, INC. Example:

```
Enter name masks or #L to list selected or #D to remove selection.
Routine name mask: MP*.MAC
Routine name mask: f*.INC
Routine name mask: #L
List of selected routine names:
MPPSRC.MAC
first.INC
-----
```

Include routines does not compiled and MAC routines are precompiled by macro preprocessor.

## 4.19 %RDELETE

Utility `^%RDELETE` removes routines and, optional, routine bytecode.

Usage

```
do ^%RDELETE
```

Utility offer to enter routines names masks. Operator can use special symbols `"?"` as pattern of any character and `"*"` as pattern of any characters sequence. If operator enter an empty string (simply press Enter or Escape), utility ends input of routines names. Sample utility screen:

```
MiniM Delete routines utility
```

```
Routine name mask: q*
Routine name mask:
```

After end of names input utility offer to select option need routines bytecode of this routines be deleted or not.

Next utility offer to enter optional file name to output report to. If operator enter an empty string, utility display report on the screen. Utility show current database name and routine names deleted with option source code or bytecode deleted. Sample utility screen:



Deletion selected routines in database USER

```
q111                (source)
q111                (bytecode)
```

After routine deletion utility ends.

If routine names are entered without extension, utility use only standard routines (INT), otherwise utility use specified routine types. In the routine extension cannot be used placeholders and utility support the following types: INT, MAC, INC. Example:

```
Enter name masks or #L to list selected or #D to remove selection.
Routine name mask: MP*.MAC
Routine name mask: f*.INC
Routine name mask: #L
List of selected routine names:
MPPSRC.MAC
first.INC
-----
```

## 4.20 %RDIR

Utility ^%RDIR show routine names are available using selected routine masks.

Usage

```
do ^%RDIR
```

Utility offer to choose routine name masks. Operator can use special symbols "?" as a pattern of one character and "\*" as a pattern of any characters sequence. If operator enter an empty string (simply press Enter or Escape), utility stops to enter names masks and offer to enter optional file name to make report to. Sample utility screen:

MiniM Routines listing utility

```
Routine name mask: *
Routine name mask:
Enter file name to print to (optional):
```

If operator enter an empty string, utility display routine names are available in the database to the screen, otherwise outputs names to file specified. Sample utility screen:

List of selected routine's in database USER

```
PITON
def
job
label
matrix
q111
```

After display routine names utility ends.

If routine names are entered without extension, utility use only standard routines (INT), otherwise utility use specified routine types. In the routine extension cannot be used placeholders and utility support the following types: INT, MAC, INC. Example:

```
Enter name masks or #L to list selected or #D to remove selection.
Routine name mask: MP*.MAC
Routine name mask: f*.INC
Routine name mask: #L
List of selected routine names:
MPPSRC.MAC
first.INC
-----
```

## 4.21 %RESTART

Utility ^%RESTART is intended to stop the server and run again.

Usage

```
do ^%RESTART
```

After run utility offer to confirm MiniM Database Server restart.

```
USER>d ^%RESTART
MiniM server restart utility.
```

Are you sure to restart server? [Y/N]:

If operator choose answer, utility exits. If operator choose "Y", utility send signal to server to stop working and server terminates all processes including current process. Next process work interrupts. Time to execute restart depends of volume of data need to be flushed to datafiles.

After stop server starts again. New server environment can differs from environment before start because server accepts any changes made in configuration files.

Internal subroutine to restart MiniM Database Server:

```
d restart ^%RESTART
```

## 4.22 %RESTORE

Utility ^%RESTORE is intended to restore database or several databases from selected backup file and show information about backup file.

Usage

```
do ^%RESTORE
```

On running utility offer to select operation type - show backup information or restore databases from backup file.

MiniM Database restore utility

Select restore option:

- 1) View backup info
- 2) View database list in backup file
- 3) Restore database from backup file

Select option:

If operator select unsupported option, utility stop working. On selection option 1) *View backup info* utility offer to enter file name of backup file and lists all available information.

Enter backup file name: full.bak

Backup info:

Full backup

File has 2 backed up databases

Backup date 03.06.2009

Next utility again offer to enter option to do.

If operator select alternative 2) *View database list in backup file* utility offer to enter backup file name and lists all available in this file backed up databases and offer to select option to do again.

Enter backup file name: full.bak

Backed up database list:

- 1) USER
- 2) %SYS

If operator select alternative 3) *Restore database from backup file* utility offer to enter backup file name and offer to confirm backup type

Enter backup file name: full.bak

Select restore type:

- 1) Full
- 2) Differential

Select option:

Next utility offer to choose journal operation to execute after database restore.

Select journal operation:

- 1) Apply journal records after backup point
- 2) Rollback uncomplete transactions from backup point
- 3) Skip journal operations

Select option:

If operator choose option 1) utility after restore search in journal appropriate backup point and applies to databases all journaled operations. If operator choose option 2) utility search in journal appropriate backup point and rolls back all transactions not committed at this backup point. And if operator choose option 3) utility does not use journal.

Next utility offer to select reporting option.

Report option:

- 1) Make report
- 2) No report

Select option:

If operator select option 1) utility offer to enter file name for report. If operator enter empty string, utility write report to current screen.

After all options selected utility restore databases from backup file. Example restore report seems like this:

```
Checking backup file full.bak
Disabling job starting and freezing active jobs...
Checking job transaction states.
Database to restore from backup: USER
Database to restore from backup: %SYS
Start restoring from backup file full.bak
Start journal phase.
Journal phase skipped.
Enabling job starting and unfreezing active jobs.
Database restore completed successfully.
```

If utility detect an error while databases restored, utility write message into report target and into screen.

On database restore utility can temporary stop other processes and can decline from restoring if one or more processes still in transactions to prevent transactions or processes states and prevent data lost.

Time to work is dependent of count of database to restore, of current datafiles state and of number of pages need to be restored (volume of backup file). Restore utility can grows and decrease databases datafiles depending of backed up database states.

Databases are restored in current database configuration of root and extent datafiles, independently of extent configuration have been used in backup.

## 4.23 %RFIND

Utility ^%RFIND execute text search in routine's source code and show routine names and lines found.

Usage

```
do ^%RFIND
```

Utility offer to enter string to find and search options - case sensitive or not.

```

USER>d ^%RFIND
MiniM Search in routines utility
Search for: *27
Case sensitive search? [Y/N]:

```

Next utility offer to enter routine names mask to search text in. Operator can use special symbols "?" as a pattern of one character and "\*" as a pattern of any characters sequence. If operator enter an empty string (simply press Enter or Escape), utility stops to enter names masks.

Next utility offer to enter optional file name to output lines found to. If operator enter an empty string, utility outputs lines found to screen.

For each existing routine with name conformed to one of masks been entered utility search line in routine source code with options specified. If this line is found, utility display routine name, label, offset and line of code.

After full search utility ends.

If routine names are entered without extension, utility use only standard routines (INT), otherwise utility use specified routine types. In the routine extension cannot be used placeholders and utility support the following types: INT, MAC, INC. Example:

```

Enter name masks or #L to list selected or #D to remove selection.
Routine name mask: MP*.MAC
Routine name mask: f*.INC
Routine name mask: #L
List of selected routine names:
MPPSRC.MAC
first.INC
-----

```

## 4.24 %RFIRST

Utility ^%RFIRST show first routine lines.

Usage

```
do ^%RFIRST
```

Utility offer to enter routine names masks to show first lines.

```
USER>d ^%RFIRST
MiniM Show routine's first lines utility
```

```
Routine name mask: *
Routine name mask:
```

If operator enter empty string (simply press Enter or Escape), utility ends query routine names masks. Operator can use special symbols "?" as a pattern of one character and "\*" as a pattern of any characters sequence.

Next utility offer to enter optional file name to output first routines lines. If operator enter an empty string, utility outputs first lines to screen.

Sample utility screen:

```

                First lines of selected routines in USER

job                job ; k d ^job w
label              label ; k d ^label w
matrix             matrix ; k d ^matrix w
```

After show first lines utility ends.

If routine names are entered without extension, utility use only standard routines (INT), otherwise utility use specified routine types. In the routine extension cannot be used placeholders and utility support the following types: INT, MAC, INC. Example:

```
Enter name masks or #L to list selected or #D to remove selection.
Routine name mask: MP*.MAC
Routine name mask: f*.INC
Routine name mask: #L
List of selected routine names:
MPPSRC.MAC
first.INC
-----
```

## 4.25 %RI

Utility ^%RI imports routine source code from file specified.

Usage

```
do ^%RI
```

Utility offer to enter file name to import routines from and check this file exists and can be opened to read. If operator enter an empty string (simply press Enter or Escape) utility ends.

```
USER>d ^%RI
MiniM routine import
Enter file name to import routines from:
```

Next utility show first two lines of file specified and suppose this two lines contains export information. Sample utility screen:

```
Enter file name to import routines from:
w:\minim\routines\misc\matrix.rtn
File w:\minim\routines\misc\matrix.rtn has been >>
  >> written with description:
9:10  10-???-2009~Format=ANSI.S~
File timestamp:
```

In depend of used export tool and MUMPS implementation and version first two lines can differs.

Next utility offer to choose what action must be performed:

```
Routine input option ([L]ist,[Q]uit,[C]ompile):
```

If operator choose command "Q", utility ends working. If have been choosen command "L", utility lists all routines names places in this file. And if have been choosen command "C", utility imports routines source code and compile all imported routine one-by-one.

Utility show detailed report what happens, sample utility screen:

```
Routine input option ([L]ist,[Q]uit,[C]ompile): c
Load matrix... compile... success
```

Utility ^%RI imports only standard routines (INT). To import macro routines (MAC, INC) use MiniM Control Center or MiniM Routine Editor or ^%RSAIN utility.

Different MiniM versions can contain small changes of reporting details.



## 4.26 %RIMF

Utility ^%RIMF imports routine bytecode from file.

Usage

```
do ^%RIMF
```

Utility offer to enter file name to import routine bytecode. One file can contain several bytecodes.

```
USER>d ^%RIMF
MiniM routine's bytecode import
Enter file name to import from:
```

If operator enter an empty string, utility ens working. Otherwise offer to choose action - chow export header from file, execute import of bytecode and show all routine names contained in this file. Sample utility screen to show header:

```
USER>d ^%RIMF
MiniM routine's bytecode import
Enter file name to import from:
w:\minim\routines\misc\matrix.mmo
Enter option ([H]eader, [I]mport, [L]ist):h
File w:\minim\routines\misc\matrix.mmo exported with header:
Matrix screen
```

Sample utility screen to show routine names:

```
USER>d ^%RIMF
MiniM routine's bytecode import
Enter file name to import from:
w:\minim\routines\misc\matrix.mmo
Enter option ([H]eader, [I]mport, [L]ist):l
matrix
```

If operator choose import bytecodes, utility execute import and replaces existing or adds unexisting bytecode to database.

MiniM processes uses bytecode to execute routine. Bytecode can differs from current routine source code, and processes does not requires to use

routines source code except the *\$text()* function. If function *\$text()* call to line of code with double comment (*;;*), source code does not required, because this lines MiniM store in bytecode.

Other MiniM instance can import this exported bytecodes and execute without routine source code. Exported bytecode can be used for MiniM servers only and MiniM cannot use bytecode of other MUMPS systems.

Next utility ends working. Time to work is depended of total bytecodes volume need to be imported.

## 4.27 %RL

Utility `^%RL` imports routine source code from file specified.

Usage

```
do ^%RL
```

Utility `^%RL` is a wrapper to `^%RI` utility for compatability with widely used names in other MUMPS systems.

## 4.28 %RO

Utility `^%RO` exports routine source code to file.

Usage

```
do ^%RO
```

Utility offer to enter routines names mask for export. Operator can use special symbols `"?"` as pattern of any character and `"?"` as pattern of any characters sequence. If operator enter an empty string (simply press Enter or Escape), utility ends input of routines names. Sample utility screen:

```
USER>d ^%RO
```

```
MiniM routine export
```

```
Enter name masks or #L to list selected or #D to remove selection.
```

```
Routine name mask: ma*
```

```
Routine name mask:
```

Next utility offer to enter file name for routine export. One file can contain several routines. File store routines source code as is, without encoding or compression.

Next utility offer to enter export description string or indicator to make autoload header.

Enter description or ^ to make auto-loading file:

Description can be any string, this string is stored in the file and can describe what this file contain.

If operator enter autoload header indicator ("^") utility creates special file header, which make possible import this file later using command line in batch mode:

```
minim.exe < filename.rou
```

Utility exports source code in ANSI format. After export utility ends. Time to work is depended of total number of routines selected for export. Utility does not lock or block routines source code and other processes can change source code in the same time.

Utility ^%RO exports only standard routines (INT). For extended export of macro routines (MAC, INC) use utility ^%RSAOUT or MiniM Control Center or MiniM Routine Editor and select Cache export format.

## 4.29 %ROMF

Utility ^%ROMF exports routine's bytecode to file, which can be imported later.

Usage

```
do ^%ROMF
```

Utility offer to select routine names masks, which bytecode need to be export. Operator can use special symbols "?" as a pattern for one character and "\*" as a pattern of any characters sequence. If operator enter an empty string, utility ends input routine names. Sample screen while utility works:

```

USER>d ^%ROMF
MiniM routine's bytecode export
Enter routine's name mask: ma*
Enter routine's name mask:

```

Next utility offer to enter file name to export bytecode. MiniM does not limit operator to use special file extensions, but operator can consider that MiniM Control Center and other GUI MiniM utilities use by default extension mmo.

Next utility offer to enter file header. This is any string been written to a file and can contain any information about export.

Next utility exports selected bytecodes to file specified. Sample screen of export:

```

USER>d ^%ROMF
MiniM routine's bytecode export
Enter routine's name mask: ma*
Enter routine's name mask:
Enter file name to export to:
w:\minim\routines\misc\matrix.mmo
Enter file comment: Matrix screen
Exported 1 routine's bytecode.

```

Next utility ends working. Time to work is depended of total bytecodes volume need to be exported.

MiniM processes uses bytecode to execute routine. Bytecode can differs from current routine source code, and processes does not requires to use routines source code except the *\$text()* function. If function *\$text()* call to line of code with double comment (;), source code does not required, because this lines MiniM store in bytecode.

Other MiniM instance can import this exported bytecodes and execute without routine source code. Exported bytecode can be used for MiniM servers only and MiniM cannot use bytecode of other MUMPS systems.

### 4.30 %RS

Utility ^%RS exports routine source code to file.

Usage

do ^%RS

Utility ^%RS is a wrapper to ^%RO utility for compatibility with widely used names in other MUMPS systems.

## 4.31 %RSAIN

Utility ^%RSAIN imports routine source code from file in the RSA format.

Usage

do ^%RSAIN

RSA file format allow to save and distinguish routine types (INC, MAC and INT) and save modification timestamp.

Utility offer to enter file name to import routines from and check this file exists and can be opened to read. If operator enter an empty string (simply press Enter or Escape) utility ends.

```
USER>d ^%RSAIN
MiniM RSA format routine import
Enter file name to import routines from:
```

Next utility show first two lines of file specified and suppose this two lines contains export information. Sample utility screen:

```
Enter file name to import routines from:
w:\minim\st\routines\test.rsa
File w:\minim\st\routines\test.rsa has >>
  >>been written with description:
this is description
File timestamp: MiniMRE on jun 09 2011 23:47
```

Next utility offer to choose what action must be performed:

Routine input option ([L]ist,[Q]uit,[C]ompile):

If operator choose command "Q", utility ends working. If have been choosen command "L", utility lists all routines names places in this file. And if have been choosen command "C", utility imports routines source code and compile all imported routine one-by-one.

Additionally utility query report file name. It is very useful if file contains many routines and compilation report can contain many error. If operator does not enter report file name, utility outputs report to current screen.

Utility show detailed report what happens, sample utility screen:

```
Routine input option ([L]ist,[Q]uit,[C]ompile): c
Enter file name for report:

Load %BACKUP.INT...
Load %CONX364.INT...

Compile %BACKUP.INT
Compile %CONX364.INT
```

On import utility first imports all routines are present in this file and at the second step recompiles all ones. Routines with the type INC does not compiled. Routines with type MAC before compilation are processed by macro preprocessor and result INT codes are compiled. Routines with type INT are compiled as is. If utility detects both routine of MAC and INT type and the same name are present, utility imports both routines but compile only MAC routine. If routine preprocessing requires that one or more routines be already compiled and must be executed on preprocessing stage, this routine must be imported before from separate file.

Different MiniM versions can contain small changes of reporting details.

## 4.32 %RSAOUT

Utility ^%RSAOUT exports routine source code to file in RSA format. RSA file format allow to save and distinguish routine types (INC, MAC and INT) and save modification timestamp.

Usage

```
do ^%RSAOUT
```

Utility offer to enter routines names mask for export. Operator can use special symbols "?" as pattern of any character and "??" as pattern of any characters sequence. If operator enter an empty string (simply press Enter or Escape), utility ends input of routines names. Sample utility screen:

```
USER>do ^%RSAOUT
MiniM RSA format routine export
```

```
Enter name masks or #L to list selected or #D to remove selection.
Routine name mask: A4*.MAC
Routine name mask:
```

Next utility offer to enter export description string. It is additional information for operator.

```
USER>do ^%RSAOUT
MiniM RSA format routine export
```

```
Enter name masks or #L to list selected or #D to remove selection.
Routine name mask: A4*.MAC
Routine name mask:
Enter description for export:
```

Description can be any string, this string is stored in the file and can describe what this file contain.

Next utility offer to enter file name for routine export. One file can contain several routines. File store routines source code as is, without encoding or compression.

```
USER>do ^%RSAOUT
MiniM RSA format routine export
```

```
Enter name masks or #L to list selected or #D to remove selection.
Routine name mask: A4*.MAC
Routine name mask:
Enter description for export:
Enter file name to export routines:
```

If operator enter empty file name, utility ends.

Utility exports source code in RSA file format. After export utility ends. Time to work is depended of total number of routines selected for export. Utility does not lock or block routines source code and other processes can change source code in the same time.

### 4.33 %SHUTDOWN

Utility ^%SHUTDOWN perform MiniM Database Server shutdown.

Usage

```
do ^%SHUTDOWN
```

On running utility offer to confirm server shutdown.

```
USER>d ^%SHUTDOWN
```

```
MiniM server shutdown utility.
```

```
Are you sure to shutdown server? [Y/N]:
```

On input selection utility stop working. If operator choose *Y* utility send internal signal to stop server working and server stops all active processes, including current process of utility and stops daemons. Next process working stops. Time to stop is dependent of volume of actions required to stop all processes and daemons.

Internal program point to stop server

```
d stop^%SHUTDOWN
```



# Chapter 5

## Macro Preprocessor

### 5.1 Macro Routines

MiniM Database Server since version 1.6 supports macro preprocessor for macro routines. Standard routines which are defined in the MUMPS language counts as INTermediate routines, routines with macros - MACro routines and routine for inclusion on preprocessing - INClude routines.

Routine names in MiniM Routine Editor and in MiniM Control Center are distinguished by logical extension, for example:

```
ROUNAME.MAC  
ROUNAME.INC  
ROUNAME.INT
```

MiniM Database Server supports the following rules: 1) INC routines does not compile into executable bytecode, 2) INT routines compile into executable bytecode and 3) MAC routines translate by macro preprocessor into appropriate INT routine with the same name and next compile into executable bytecode.

Macro routines MAC and INC which start from the percent symbol (%) are mapped using the same rules as standard INT routines, are stored in the %SYS database and are visible from any other database.

Macro routines are the source code for the standard INT routines and following executable bytecode. Macro preprocessor scans line-by-line sequentially and if line contains macro directive execute this directive or if contains macros, substitutes this. Macro routine can contain simple lines

with the MUMPS code, macro directives, macros or any combination by the programmer opinion.

Different routine types can have the same names. The routine source code is stored in different globals. After MAC routine translation result of INT routine is available for viewing or editing.

MiniM supports macros only in the macro routines. Indirection, *xecute* commands, command line mode and MWA tags does not supports macros.

MiniM Routine Editor supports syntax highlighting for macro directives and macros.

MiniM Routine Editor and MiniM Control Center supports extended format for export and import routines in Cache RSA format (Routine Save Archive) with differentiating by routine types and saves the change timestamp.

## 5.2 #define

Directive `#define` defines the name of macros with optional arguments and optional substitution string. Name must have first symbol letter and can contain letters and digits.

Directive has two arguments - macros name and substitution string:

```
#define NAME[(args)] [string for replace]
```

Substitution string is optional, and without one macros replaces to an empty string. Macros arguments are optional too. If arguments are present, they must be specified inside parenthesis after macros name and each argument must have leading symbols percent (%) and letter, and following letters and digits. For example:

```
#define NAME(%arg1) ^ABC("NAME",%arg1)
```

On macros usage must be specified leading symbols \$\$\$\$. For example:

```
write $$$NAME(123)
write $d($$$$NAME(456))
```

translates to INT routine as lines:

```
write ^ABC("NAME",123)
write $d(^ABC("NAME",456))
```

If macros have an arguments, on macros usage arguments must be specified too.

On substitution macros arguments substitutes by names, For example if we have a macros:

```
#define NAME(%n,%v) s ^ABC("ind",%n)=%v
```

than on usage

```
$$$NAME(123,456)
```

this macros translates into

```
s ^ABC("ind",123)=456
```

Macros allows enclosure, macros can be used as an arguments or a part of other macros. For example, the following text:

```
#define A1 "A"
#define B1(%a) BB(%a)
w $$$B1($$$A1)
```

translates into

```
w BB("A")
```

If macros is present inside the string or inside the comment, this macros does not substitutes, for example:

```
#define A1 "A"
#define B1(%a) BB(%a)
w "$$$B1($$$A1)"
; $$$B1($$$A1)
```

translates into

```
w "$$$B1($$$A1)"
; $$$B1($$$A1)
```

Directive `#define` must use the entire line of code. On translation no line produces to INT routine.

After macros been defined this name counts as defined name for preprocessor independently of macros arguments and existence of defined macros can be checked by directives `#ifdef` or `#ifndef`.

After macros been defined this name redefinition counts as an error independently of macros arguments.

If macros have not been defined, this name cannot be use and name usage counts as a preprocessor error.

Macros definition can be removed by the preprocessor directive `#undef`.

### 5.3 Macro Comment

Macro comment starts from the following symbols:

```
#;
```

and after ones follows the comment text. On the preprocessing the entire line does not includes into output INT routine.

Macro comment must be started from the line begin. If before symbol `#` are present any nonspace symbols, entire macro comment inserted by preprocessor into output INT routine and give syntax error on INT routine compilation into bytecode.

Example:

```
;; comment included into bytecode  
;; and into INT routine  
  
; comment included into INT routine  
; but not into bytecode  
  
#; comment does not included  
#; into INT routine or bytecode
```

## 5.4 #else

Directive #else switches condition of conditional preprocessor block. This directive applies to prior directive #if or #ifdef or #ifndef.

Directive #else splits conditional block to two parts. If block condition evaluates as true, preprocessor use lines from start directive #if or #ifdef or #ifndef up to #else directive, otherwise preprocessor use lines from the #else directive to end of block, directive #endif.

Example:

```
#if $zv["MiniM"  
  #; code for MiniM  
  ...  
#else  
  #; code for others  
  ...  
#endif
```

Directives #if, #ifdef and #ifndef allows inclusion to each other and preprocessor use pairs of block start and directives #else and #endif.

## 5.5 #endif

Directive #endif terminates conditional block of lines, from #if or #ifdef or #ifndef directive.

On this directive translation preprocessor removes current processing condition level and continues processing on previous level.

Example how to terminate conditional block with #else alternative:

```
#if $zv["MiniM"  
  #; code for MiniM  
  ...  
#else  
  #; code for others  
  ...  
#endif
```

Example how to terminate block started with directive #ifndef:

```

    #; application definitions
#ifndef incDEFINESINCLUDED
#define incDEFINESINCLUDED

#define APPDATA(%id) ^APP("data",%id)
#define APPIND(%n,%v) ^APP("ind",%n,%v)

#endif

```

Directives `#if`, `#ifdef` and `#ifndef` allows inclusion to each other and preprocessor use pairs of block start and directives `#else` and `#endif`.

## 5.6 `#execute`

Preprocessor directive `#execute` executes argument as line of MUMPS commands. Commands executes by the *xecute* command as is.

MiniM macro preprocessor use local variables with the `%mpp` prefix. Don't use this variables, this can break preprocessor.

As an argument of the `#execute` directive can be used any commands, wich are allowed as a *xecute* arguments, and can be used output to the current device. It is not recommended to use read commands from current device, because macro translation context does not suppose this input and any other interaction with the user.

Programmer can use side effects of directive `#execute` and create sequence of this directives and combine side effects of directives `#execute` and `#if`.

```

:
#if $zv["MiniM"
    #; terminate this job
    k ^$JOB(pid)
#else
#execute w "WARNING: Unknown job termination.",!
#endif

```

If directives `#if` or `#execute` have side effects as current device changes, it is required to restore current device to continue normal preprocessor execution.

## 5.7 #if

Directive `#if` starts conditional block of lines. Directive must have mandatory argument as expression on the MUMPS language.

Directive `#if` starts conditional block and block of lines counts as a line's sequence up to next pair directive `#endif`. This conditional block can be splitted by two parts by the `#else` directive.

On directive processing preprocessor evaluates argument as an expression and if result is nonzero, preprocessor includes lines up to next pair directive `#else` or `#endif`. If result evaluates as zero, next lines does not includes but if next pair directive is a `#else` directive, lines includes from `#else` up to `#endif`.

Example:

```
#if $r(2)
  w 123,!
#endif
```

Here preprocessor evaluates value of function `$r(2)` and in depend of the result includes or not line with the *write* command.

Directives `#if`, `#ifdef` and `#ifndef` allows inclusion to each other and preprocessor use pairs of block start and directives `#else` and `#endif`.

Directive `#if` unlike of the *if* command evaluates once and on the compiling stage. This directive can be used to generate code in dependent of current MUMPS environment:

```
#if $zv["MiniM"
  #; code for MiniM
  ...
#endif
#if $zv["Cache"
  #; code for Cache
  ...
#endif
```

After compilation of this MAC routine in MiniM we got INT routine with special code for MiniM, if compiled in Cache, we got code for Cache and others.

Preprocessor use internal local variables with the `%mpp` prefix. Don't use this variables, this can break preprocessor. Programmer can combine side effects of the `#if` and `#execute` directives.

## 5.8 #ifdef

Preprocessor directive `#ifdef` starts conditional block of lines. Directive must have one mandatory argument as macros name without arguments.

Directive `#ifdef` starts conditional block which starts on the next line and continues up to next pair directive `#endif`. Block can be splitted by two parts by the `#else` directive.

Directive checks this macros been defined. Name specified can have or not substitution string. If this name have been defined previously, preprocessor includes following lines after `#ifdef` directive and up to next pair `#else` of `#endif` directive. Otherwise preprocessor skip lines up to next `#endif` directive and if reacts pair `#else` directive, continues processing lines up to next pair `#endif` directive.

Example:

```
#if $zv["MiniM"  
    #define verMiniM  
#endif  
...  
#ifdef verMiniM  
    ; code for MiniM  
    ...  
#else  
    ; code for other MUMPS  
    ...  
#endif
```

Directives `#if`, `#ifdef` and `#ifndef` allows inclusion to each other and preprocessor use pairs of block start and directives `#else` and `#endif`.

## 5.9 #ifndef

Preprocessor directive `#ifndef` starts conditional block of lines. Directive must have one mandatory argument as macros name without arguments.

Directive `#ifndef` starts conditional block which starts on the next line and continues up to next pair directive `#endif`. Block can be splitted by two parts by the `#else` directive.



Directive checks this macros has not been defined. Name specified can have or not substitution string. If this name have not been defined previously, preprocessor includes following lines after #ifndef directive and up to next pair #else of #endif directive. Otherwise preprocessor skip lines up to next #endif directive and if reacts pair #else directive, continues processing lines up to next pair #endif directive.

Example of include file defines.INC:

```

#; application definitions
#ifndef incDEFINESINCLUDED
#define incDEFINESINCLUDED

#define APPDATA(%id) ^APP("data",%id)
#define APPIND(%n,%v) ^APP("ind",%n,%v)

#endif

```

Directives #if, #ifdef and #ifndef allows inclusion to each other and preprocessor use pairs of block start and directives #else and #endif.

## 5.10 #include

Preprocessor directive #include includes specified in the argument INC routine. The name of the INC routine must be specified without type or extension.

Example:

```

#include common
#include defines

```

This code includes INC routines common.INC and defines.INC.

While directive #include processed, preprocessor includes instead this line all lines of specified INC routine. The INC routine content can be any macro code, including subroutines on MUMPS language, preprocessor directives or macros usage.

MiniM Database Server does not compile INC routines into intermediate routines, this routines intended only for inclusion by preprocessor. INC routines can have the same names as MAC or INT routines.

MiniM macro preprocessor allow secondary INC routine inclusion, but contains protection from inclusion recursion. In the case of INC routine contains and must allow recursion, this routine must contain protection from recursion and from macros redefinition with macros name derives from the INC routine name, for example:

```

    #; application definitions
#ifndef incDEFINESINCLUDED
#define incDEFINESINCLUDED

    #define APPDATA(%id) ^APP("data",%id)
    #define APPIND(%n,%v) ^APP("ind",%n,%v)

#endif

```

If preprocessor detect inclusion resursion of INC routine, preprocessor raise once error and does not execute recursion.

## 5.11 #undef

Preprocessor directive #undef removes definition of the macros specified. Directive must have one mandatory argument as macros name without arguments.

This directive removes macros with this name from internal preprocessor definitions. Name must be specified without arguments. Directive removes macros definition independently of the macros arguments. If this macros name have not been defined, directive have not side effects.

Example:

```

#define NAME abc
s $$$NAME=123
#undef NAME
#define NAME def
s $$$NAME=456

```

Here directive #undef removes definition of the NAME macros and the following macros definition does not raise preprocessor error about macros redefinition.

## 5.12 Macro functions

MiniM Database Server since version 1.12 implements macro functions, which may significantly extend user-defined macro and macro expressions.

Macro functions can be called in context of evaluable macro directives `#if` and `#execute`. In this context preprocessor routine `%MPP` is current and macro functions implemented in routine `%MPP` may be used without routine name specification.

Macro functions can change macro substitution, check macro existence, return value, and other. In context of evaluable macro directives macro functions usage can extend developer possibilities, for example, developer can create logical conditions over macros, generate macro substitution, pre-evaluate macro expression at compile time and other.

### **DEFINED(mname)**

Macro function `DEFINED` returns value 1 if macro with name `mname` was defined and value 0 in other case. Macro name must be specified in double quotes by MUMPS language rules. Developer can use macro name evaluation too.

Macro function `DEFINED` allow to create logical expressions at compile time in depends of defined macro or not. For example:

```
#define MACRO1

#if $$DEFINED("MACRO1")&$$DEFINED("MACRO2")
  w "both macro1 and macro2 defined"
#else
  w "one of macro1 or macro2 not defined"
#endif

#define MACRO2

#if $$DEFINED("MACRO1")&$$DEFINED("MACRO2")
  w "both macro1 and macro2 defined"
#else
  w "one of macro1 or macro2 not defined"
#endif
```

This code generate the following INT code:

```
w "one of macro1 or macro2 not defined"
w "both macro1 and macro2 defined"
```

### **GET(mname,default="")**

Macro function GET returns current macro substitution of mname macro or return the default value if macro was not defined or does not have substitution.

For example, routine:

```
#define APPVERSION 3
; ...
#if $$GET("APPVERSION")>2
w "Code for application version greater than 2"
#else
w "Code for compatibility with version 1"
#endif
```

generates the code in dependence of application version number.

### **QUOTE(str)**

Macro function QUOTE returns argument, decorated by MUMPS language rules as a printable and readable string. For example, routine:

```
#execute d SET("LIST",$$QUOTE($1b(12,34,56)))
w $$$LIST
```

will evaluate at compile time some (may be very hard) expression and transforms result to a MUMPS string:

```
w $C(3,4,12,3,4)_"_"_$C(3,4)_"8"
```

In this example, the complexity of an expression is very relative, but real applications can significantly increase performance if preevaluate some expressions.

### **SET(mname,msubst)**

Macro function SET creates or replaces value of macro substitution. Substitution value evaluates at compile time. For example, macro routine:

```
w "Routine executed at "_$zd($h,3)
#execute d SET("COMPILED",$$QUOTE($zd($h,3)))
w "Routine compiled at "$$$COMPILED
```

will generate the code like this

```
w "Routine executed at "_$zd($h,3)
w "Routine compiled at "_"2011-12-25"
```

Here first line outputs execution time, and second outputs compilation time. Developers can preevaluate some set of constants, use owned substitution generation functions.

### **EVAL(expr)**

Macro function, reverse to macro function QUOTE, evaluates expression was specified by MUMPS language rules. For example, the following macro code

```
#define MACROEXPR 1+2+3
#execute d SET("EXPAND",$$EVAL($$GET("MACROEXPR")))
w $$$EXPAND
```

will generate INT code with substitution of evaluated macro:

```
w 6
```

This example defines macro MACROEXPR as a string 1+2+3, next line evaluates value of substitution (GET("MACROEXPR")), evaluates as a MUMPS expression (EVAL), and result writes as a substitution to a macro EXPAND. In result ordinal macrocode can use macro EXPAND with evaluation result of MACROEXPR.

In context of evaluatable macro directives #if and #execute developer can use complex combinations of macro functions and call own functions and substitution generators.



# Chapter 6

## MiniMono

### 6.1 MiniMono Architecture

MiniMono is a short name of special single-user MiniM edition, full name is MiniM Embedded Edition. MiniM Embedded Edition is a dynamic link library with MiniM modules and intended to use within personal applications to embed into application MUMPS language and database possibilities and functionality.

Now MiniMono is supported for the same operating systems as full MiniM Database Server and have examples how to use for all supported operating systems.

Application creates MiniMono virtual machine, call him to execute MUMPS commands, evaluate expression values or change variables values. In a host application context MiniMono works as a MiniM job and some write daemons. MiniMono creates a daemon child threads as need. MiniMono virtual machine can work with one database stored in the same format as such as in full MiniM Database Server without no difference.

While MiniMono is active, application can use all MiniM Database Server possibilities except to run child jobs and use multifile databases.

Full MiniM Database Server contains special things to improve data stability and full server works with operating system stability plus database restoring by before image journal and restoring from backup. MiniMono as a simple library is limited by host process stability and works with application stability plus database restoring by before image journal and restoring from backup.

MiniMono contains code for database expand daemon, write daemon and journal daemon. Host application developer must understand that full MiniM

Database Server contains special protection things as process guardians and MiniMono threads does not contain them and host application developer can use only database restoring by before image journal and from backup. As such as full MiniM Database Server, MiniMono contains hot backup and restore too.

Common MiniMono use sequence is the following: 1) get default initialization values for virtual machine, 2) change default values as need by application, 3) create virtual machine, 4) call MiniMono and 5) terminate MiniMono virtual machine.

MiniMono does not store own settings in any INI files or Windows registry. It is considered that host application store own settings as need. One MiniMono library can serve several host applications if ones use different databases and does not allow to use one database by different host applications at one time. Each MiniMono virtual machine contains own MiniM modules including own globals and routines caches and write daemons, so several host applications cannot work with one database to prevent database corruption.

One host application can create only one MiniMono virtual machine. Host application developer must understand that in his application address space will present several secondary MiniMono objects and with incorrectly written application the host process can corrupt internal MiniMono data structures.

MiniMono does not implement own user interface, character oriented or windowed. MiniMono virtual machine by default creates special virtual input output device DLL, and host application should define event handlers for this device. Host application can be console oriented or windowed or other and must define this DLL device behavior self.

All data and routines of MiniMono are stored in a single database and MiniMono works with database as a MiniM with the "%SYS" database. So, common recommendations can include installing MiniM Database Server for special instrumental purpose to prepare the "%SYS" database and use prepared database next with a target host application. Host application developer can change system routines set as need or does not use any routines.

MiniMono does not support MiniM debugger, so if developer want to debug MUMPS routines, he must use special MiniM Database Server instance and import debugged routines into MiniMono database.

Functions and behavior of MiniM Embedded Edition is the same as in the full MiniM Database Server of the same version. As such as within MiniM Database Server, MiniMono can use the same ZDLL and ZDEVICE



extensions, system functions, transactions, system and user-defined routines. MiniMono contains some limitation, listed in special topic "Difference list".

MiniMono installs with samples, and with utilities:

minimonostd.exe	Command-line utility to get input from stdin and output data to stdout
minimonocon.exe	Console utility, gets input from keyboard and outputs data to console with processing escape sequences
minimonore.exe	MiniMono Routine Editor, utility to edit, compile, export and import routines, bytetimes and globals
minimonoge.exe	MiniMono Global Editor, utility to view and edit globals
minimne.exe	Utility to edit files with character collation definitions

MiniMono uses the same documentation as such as full MiniM Database Server.

To install MiniMono with target host application it is enough to copy to target computer files minimono.dll into the directory which is accessible for host application. It is not need any other installation steps or registrations.

## 6.2 Data structures

MiniMono use the same data structures as such as ZDLL modules and the same calling conventions and data encodings.

Main data transfer structure is a MINIM\_STR structure. This structure can contain an ordinal byte sequence as such as numbers. The data type have been places is determined by special field len or type. Host application can allocate memory only for really used length, for example only for 100 bytes plus length for special field len or type.

On send and receive data to and from MiniMono context host application must create appropriate data structures with need memory class. On call back host application MiniMono virtual machine send pointers to own internal data structures.

To interact with MiniMono virtual machine host application use the same data structure as such as ZDLL modules, it is ZDLLCB structure. MiniMono

virtual machine creates pointer to this structure on creation. Structure contains set of pointers to functions to call MiniMono context. For MiniMono it is set of direct call functions and for ZDL modules it is set of call-back functions.

For MiniMono virtual machine initialization is used MINIMONOVMM structure. This structure contains virtual machine parameters. All structure fields must be filled by host application, but cbfunc field is initialized by MiniMono virtual machine with real pointers to functions.

To simplify initialization MiniMono intend special function to fill MINIMONOVMM structure with default values. Default values was selected by typical parameters of most widely used personal application requirements. After getting default values host application must fill application-specific differences of MiniMono virtual machine parameters, real data file location and DLL device event handlers if they are present.

DLL device event handlers are fully compatible with the same event handler for ZDEVICE virtual devices with some exceptions: 1) DLL device can be only one and event handlers have not any device instance context and 2) device have not open and close event handlers.

MINIMONOVMM fields list:

DataFile	Database file name, can be specified full or relative path. Default value NULL, must be set by host application
ReadOnly	Use database in read only (1) mode or write is allowed (0). If database used in read only mode, MiniMono does not run write daemons. Default value 0, write enabled.
JournalingEnabled	Enabled (1) or disabled (0) database journaling. This field used id database write was allowed. If journaling was disabled (0), MiniMono does not run journal daemon and cannot roll back globals changes on rollback transactions. Default value 1, journaling enabled.

LockAreaSize	Size of locking table for lock commands in megabytes. While MiniMono works in single-user mode, application developer can estimate lock table size by his application requirements. In single user mode locking does not required, but can be used because commands can be present in routines. If locking table space overflows, process still wait free space, but no other process can remove locks, so this space must be enough. Default value 1 MB.
RoutineCacheSize	Size of bytecode cache in megabytes. If routine use small number of routines, host application developer can specify small value, and if application use many routines, increasing this value can increase performance. Default value 1 MB.
DeviceTableSize	Number of input-output devices can be created by MiniMono including default DLL device. Default value 4.
DeviceNameSize	Maximum length of device to distinguish device names. Default value 400 bytes.
DBCacheSize	Global's cache size in megabytes. It is recommended value of cache in consider that MiniMono virtual machine must place data structures in host application's address space and a really available operating memory on computer. If value of global cache is much more than physically available, speed can be decreased by swap operations. Default value 100 MB.
NullSubscripts	Enabled (1) or disabled (0) null subscripts in local and global variables. Default value 0, null subscripts usage raise an error.
TransactLevelLimit	Maximum number of transaction level. Default value 255.
TrapOnEof	Need (1) or not (0) generate an <ENDOFFILE> error if current device reach end of file. Default value 1, if end of file reached, MiniMono raise an error.

FrameCount	Maximum number of stack frames for subroutines and xecute commands. Default value 1024.
JournalCache	Size of journal cache in megabytes. Default value 4 MB.
LocaleFileName	Name of file of character definition (nat file) or null value to use default characters collation rules. Default value NULL, host application can set to custom collation file or use one of MiniM installed, or don't use special collation rules.
ProcessStorage	Size of local variable storage in megabytes. Default value 8MB.

Default values for DLL device event handlers are empty pointers. If host application specify some event handler, MiniMono call this functions on device operations.

### 6.3 Direct calls

Direct MiniMono virtual machine call is intended to execute MUMPS commands sequences, evaluate MUMPS expression values, data casting to strings or to numbers, global and local variables operations, and call subroutines.

Host application for each MiniMono call must create data structures, fill by data in own memory and if need, structures to receive data from MiniMono. MiniMono application interface was built without additional memory allocation-free functions and all operations are made in host application memory or in MiniMono internal data structures.

Direct MiniMono virtual machine calls are the same as ZDLL calls and return error codes have prefixes ZDLL.

To get real pointers to functions host application must create MiniMono virtual machine. This operation fills up pointers to own functions in cbfunc field of MINIMONOVm structure.

After MiniMono virtual machine terminates, host application must not call this function even pointers are not null.

## 6.4 Calls back

On direct calls execution MiniMono virtual machine executes commands and MUMPS routine can call principal device DLL. On this events MiniMono calls host application back.

On call back MiniMono virtual machine pass control to specified by host application functions. In the case of event handled does not present, MiniMono prepares default actions as such for NULL device.

All data structures passed to event handlers to host applications are internal structures of MiniMono virtual machine and host application must not write this data or make any additional decision about bytes are available or not over really used number of bytes.

All event handlers must return value of 0 on success and non 0 on error. If event handler return non 0 value, MiniMono virtual machine generates an error and MUMPS routine can catch one by error handlers.

All defined event handlers in one's part can call MiniMono context back to evaluate expression, execute commands, cast data or change local or global variables.

Event handler's behavior, what was defined, the USE command's options and EOF state handling are fully defined by host application.

Read string (type `dlldevreadstr_t`) and read character (type `dlldevreadchar_t`) gets as a parameter value of timeout in milliseconds. If MUMPS routine does not specify read timeout, MiniMono pass default value of -1. If MUMPS routine does not specify read length, MiniMono pass default value of -1.

MiniMono virtual machine assume that host application developer self creates EOF state handling and event handlers are coherent by this state. EOF get event handler can be called by MiniMono virtual machine in any context repeatedly.

DLL device events have not open and close handlers. This device is a principal input-output device and cannot be directly created or closed, so have not open and close options. Host application developer must create initialization and deinitialization code for DLL device structures before MiniMono virtual machine creation and after termination.

## 6.5 Difference list

MiniM Embedded Edition contains all functionality of the full MiniM Database Server except listed next.

MiniMono architecture does not contains special things to improve data stability and process protection (process and daemon guardians).

MiniMono does not use any external settings stored in INI files or in Windows registry, all settings host application must store and use self.

MiniMono does not implement devices with types CON, STD and TNT, because user interface host application must implement self. MiniMono does not make any limitations for application type - console, windowed or any other.

MiniMono does not implement device with MEM type, because it is interprocess-oriented device and MiniMono does not support multiple MUMPS jobs.

MiniMono automatically creates special files - log `minim.log`, before image journal `minim.bij` and journal files in the same directory where datafile is located. So, host application must place database in separate subdirectory. Different databases can be places in one directory for different applications if ones use database in read only mode.

The JOB command does not implemented because MiniMono is a single-user MiniM implementation.

MiniMono does not support MiniM debugger because MiniM debugger have multiprocess-oriented architecture.

System variable `$JOB` returns thread number of operating system instead of process number as in full MiniM Database Server.

System variable `$ZPARENT` always returns value of 0, because MiniMono job does not have any parent MUMPS job.

System variable `$ZVERSION` contains product name MiniMono instead of MiniM to distinguish job context for MUMPS routines.

MiniMono always have instance name as "MINIMONO". It is virtual instance name and this name have all MiniMono virtual machines. In the same time on the computer can be installed MiniM instance with the same name and one and MiniMono machines does not intersects by shared memory between each other.

Character definition file (collation definition) can be placed in any sub-directory and have any name, unlike of MiniM Database Server.

MiniMono supports only one database and database must have only one root datafile without extents. Database can have any size. Database always have name of "%SYS".

MiniMono does not implement automatic data mapping for globals and routines into other databases, because supports only one database.

Additionally with MiniM Database Server devices MiniMono adds one more special virtual device DLL. This device creates automatically as a principal input-output device. Host application must specify what device event handlers are implemented. For unimplemented events MiniMono use default behavior as such as for NULL device - write lost data and read return control immediately with empty data, system variables for current device returns default values. Host application can defined all events behavior entirely, MiniMono does not make any limitations.

The HALT command in MiniMono does not terminate host application, but terminates current call to MiniMono. After next call was terminated by the HALT command, host application can call MiniMono again, for example execute commands or evaluate expression.

MiniMono does not exposes virtual machine performance counters as a Windows performance counters (in edition for Windows), but all counters are still accessible via special \$view("perf") functions.

MiniMono virtual machine is a free of royalty software and does not requires any license key.

Common MiniM Language Guide contains MiniMono differences from full MiniM Database Server in appropriate guide topics.

## 6.6 MiniMono CHUI Tools

MiniMono Embedded Edition contains two CHUI utilities, minimonostd.exe and minimonocon.exe. Is is CHUI shells to MiniMono module.

minimonostd.exe utility uses standard input-output channels of console process - stdin and stdout. All input from stdin threatred as a strings and executes as a lines of MUMPS commands. All output which made by write commands redirects to stdout channel.

minimonostd.exe utility may be used in a batch files with input-output redirection.

Input-output behaviour of `minimonostd.exe` is full analog of |STD| device of full MiniM Database Server.

`minimonocon` utility (`minimonocon.exe` for Windows and `minimonocon` for Linux) uses input from the keyboard and outputs data to the console. On the output utility uses escape sequence processing. On process start current device |DLL| have mnemonic routine "%CONX364". If MUMPS commands calls mnemonics, this routine must be imported with compilation.

Input-output behaviour of `minimonocon` is full analog of |CON| device of full MiniM Database Server.

To specify process options both utilities support command-line parameters. It is need to specify database file name, startup actions and process parameters.

`-d` `datafile` option specify file name of MiniM database. This option is mandatory.

Option `-n` `natfile` specify the name of character collation file. This option is optional, but if need to edit globals with national characters, this option is much recommended to make correct global records.

`-r` `readonly` option show need or not process use this database in readonly mode. If option argument is not 0, process use database in readonly mode. By default write allowed.

`-c` `cacheinmegabytes` option specify size of database cache in megabytes. If option was not specified, process use default value.

`-j` `journaling` option allow or not database journaling. If option argument is 0, journaling is disabled. By default journaling is enabled. If process disable journaling, trollback command cannot revert database changes made by process whithin transaction.

`-p` `show prompt` option specify need or not to be shown standard MUMPS prompt. By default process show prompt, but prompt does not outputs while executes packet commands from the `-x` option.

`-x` `xecute` option specify line of MUMPS commands need to be executed on process start.

`-x @xecutefile` option specify name of file with lines of MUMPS commands to execute on process start. Lines are executed sequentially, line-by-line.

`-h` option specify that process must terminate after `-x` option executed. If `-h` option was not specified, after `-x` execution process goes to ordinal input lines of commands from appropriate input.



Example of MiniMono CHUI tools usage - create batch command file and specify need options, for example:

```
minimonostd.exe -d empty.dat
```

Or

```
start minimonocon.exe -d sys.dat -j 0 -x "d ^%aNC"
```

Unlike of full MiniM Database Server, MiniMono CHUI tools `minimonostd.exe` and `minimonocon.exe` starts execution at stack level 1, not 0, it is by design of MiniMono internal architecture.

## 6.7 MiniMono GUI Tools

MiniM Embedded Edition installation contains two GUI tools, `minimonore.exe` and `minimonoge.exe`. There are GUI shells for MiniMono.

`minimonore.exe` utility is a full-functional analog of MiniM Routine Editor, except client side of MiniM Debugger. MiniMono architecture does not implement MiniM Debugger.

All other functionality of MiniMono Routine Editor are the same as MiniM Routine Editor, including routine editing, preprocessor support (if server-side routines are imported from `install.rou`), export and import of routines and globals.

MiniMono Routine Editor supports command-line switches to specify database file and routine to open.

Option `-d datafile` specify the name of MiniM database file to use. This option is optional, and if `datafile` does not specified, utility lets to select `datafile`.

Option `-n natfile` specify the name of character collation file. This option is optional, but if need to edit globals with national characters, this option is much recommended to make correct global records.

Option `-r routine` specify the name of routine to open at start. This option is optional.

Example:

```
minimonore.exe -d sys.dat -r %BACKUP
```

minimonoge.exe utility is a MiniMono Global Editor, full-functional analog of MiniM Global Editor, but for MiniMono.

MiniMono Global Editor supports command-line switches to specify database file and global to open.

Option -d datafile specify the name of MiniM database file to use. This option is optional, and if datafile does not specified, utility lets to select datafile.

Option -g global specify the name of global to open at start of MiniMono Global Editor. This option is optional.

Option -n natfile specify the name of character collation file. This option is optional, but if need to edit globals with national characters, this option is much recommended to make correct global records.

Example:

```
minimonoge.exe -d sys.dat -g ^ROUTINE
```

## 6.8 MiniMonoX

Library MiniMonoX.dll is an ActiveX interface to the MiniMono library (MiniM Embedded Edition) and transforms calls from OLE Automation interfaces to the internal MiniMono interfaces. Before run this ActiveX component on the client computer must be placed in accessible directory both components - minimonox.dll and minimono.dll. MiniMonoX library must be registered after installation running the following commands in Windows console or inside installer:

```
regsvr32 minimonox.dll
```

In depends of installer purposes can be used different options of the regsvr32 command or special WinAPI calls for registering ActiveX component.

MiniMonoX library was made in both variants - for x86-32 and for x86-64. To use proper version MiniMonoX must be used with the same bit technology as MiniMono component.

MiniMonoX library implements two ActiveX objects:

```
MiniMono.ServerString  
MiniMono.VM
```

MiniMono.VM type was intended for connection from ActiveX interface to MiniMono virtual machine, call MiniMono and for receiving events of principal device. Internal type MiniMono.ServerString was intended for storing and transforming data between ActiveX interfaces and internal data types, used by MiniMono. Some part of MiniMono.VM object methods has understudy with Str suffix. In this case main function works with data in full format in MiniMono.ServerString object, functions with Str suffix works with ordinal OLE Automation strings (BSTR). MiniMono.ServerString objects can accept and store all bytes, including unused in ordinal strings, for example, binary data for \$Ib() structures, zero bytes, and correctly translates all data from internal representation from Unicode (for OLE Automation) to the standard ANSI sequences of bytes.

Architecture of MiniMono virtual machine make some limitation for calls from the client application - 1) MiniMono.VM object can be created only in one instance inside of client application and 2) with one datafile can work in one moment only one MiniMono instance.

#### Common scheme of work

Common scheme of work with MiniMonoX object consists of creation and removing of object MiniMono.VM, initializing by specifying of datafile, localization file, cache settings and other options, creating and terminating virtual machine of MiniMono, and calls to created virtual machine. Objects of MiniMono.ServerString type and ordinal strings are used for data transfer and transform by needs.

Example on VBS:

```
' create ActiveX object
Dim MiniMono
Set MiniMono = WScript.CreateObject( "MiniMono.VM")
' initialize virtual machine
MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0
' create virtual machine
MiniMono.CreateVM
' ...
' calls to MiniMono virtual machine
' ...
' free virtual machine
MiniMono.FreeVM
```

```
' free ActiveX object
Set MiniMono = Nothing
```

To use special behaviour of principal device there need to be defined set of principal device event handlers in host application:

```
Dim MiniMono
Set MiniMono=WScript.CreateObject("MiniMono.VM","MiniMono_")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM
MiniMono.ExecuteStr "w 123"
MiniMono.FreeVM

Set MiniMono = Nothing

Sub MiniMono_DevWriteStr( Str)
    WScript.Echo str.Value
End Sub
```

Here function `WScript.CreateObject` has been used in two-argument form with definition of prefix for event handlers. Names of event handlers in VBS hosts consists of this specified prefix (defined by developer) and name of event handler (defined by ActiveX object), for example:

```
DevWriteStr
DevWriteChar
DevWriteNL
```

Instead of name prefix in VBS hosts can be used any string, choosed by developer. Other runtime environments must use call conventions and naming conventions supported by this runtime environment.

MiniMono virtual machine can be created only once in host application and principal device can be only one, so device event handlers does not contain identification arguments of current virtual machine instance and device instance.

### 6.8.1 MiniMono.VM properties

Properties of MiniMono.VM object must be assigned between creation of object and creation of MiniMono virtual machine by this object. This properties are used for initialization of virtual machine. After creation of MiniMono virtual machine this properties are not affected, but still accessible for read and write.

Example:

```
Dim MiniMono
' create object
Set MiniMono = WScript.CreateObject( "MiniMono.VM")
' assign initial values
MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0
' create virtual machine
MiniMono.CreateVM
```

#### DataFile

Property DataFile has an ordinal string type and must point to a datafile whose must be used as a database file by MiniMono virtual machine. Path pointed to this file may be absolute as such as relative from current directory, or a datafile must be in one of directories listed in the PATH environment variable.

Property DataFile must be specified mandatory, it is minimum of information need to start virtual machine and has not default value or values.

#### LocaleFileName

Property LocaleFileName has an ordinal string type and points to localization file NAT, which need to be used for collation operations and to define which bytes are letters in upper or lower cases and which bytes are letters. This file may be specified by absolute path or relative from the current directory or must be found in directories listed in the PATH environment variable.

Property LocaleFileName is optional and by default MiniMono virtual machine uses internal default collation table.

### ReadOnly

Property ReadOnly has an integer type and if not equal by zero, MiniMono virtual machine runs in read only mode.

Property ReadOnly is optional and by default virtual machine uses datafile for reading and writing.

### JournalingEnabled

Property JournalingEnabled has an integer type and if not equal by zero, journaling of the database is disabled.

Property JournalingEnabled is optional, and by default journaling is enabled. If journaling was disabled, virtual machine does not journal any global changes and TROLLBACK command does nothing.

### LockAreaSize

Property LockAreaSize has an integer type and defines size of memory area need to be used by virtual machine to store information about locks in megabytes.

MiniMono virtual machine has special architecture for only one process, so applications was designed specially for MiniMono does nor requires locking operations by LOCK command.

Property LockAreaSize is optional, by default virtual machine use memory area by 1 megabyte, and values need to be between 1 and 64 megabytes. Minimum size of 1 megabytes need to be reserved for internal locking operations to synchronize job and internal daemons.

### RoutineCacheSize

Property RoutineCacheSize has an integre value and defines size of cache for compiled bytecode in megabytes.

Property RoutineCacheSize is optional, by default virtual machine uses 1 megabyte and value must be between of 1 and 64 megabytes.

### DeviceTableSize

Property `DeviceTableSize` has an integer type and defines how much devices can be opened by job in one time. Property is optional and must be between of 4 and 1000.

#### `DeviceNameSize`

Property `DeviceNameSize` has an integer type and defines maximum length of device name whose can use job. Property is optional, value by default is 400.

#### `DBCcacheSize`

Property `DBCcacheSize` has an integer type and defines size of memory used by global's cache in megabytes. Value by default is 100 megabytes. Minimum value is 1 megabyte. For 32-bit version of MiniMono maximum value is 1 gigabyte and for 64-bit version maximum value has no limits.

#### `NullSubscripts`

Property `NullSubscripts` has an integer type and defines allow or not empty strings as a subscript values for local and global variables. If this property has zero value, this means that empty strings does not allowed as a subscript value, otherwise allowed. By default for compatibility with legacy MUMPS systems empty strings does not allowed.

#### `TransactLevelLimit`

Property `TransactLevelLimit` has an integer type and defines maximum level of transactions in process. Value by default is 255, minimum value is 1 and maximum is 32000.

#### `TrapOnEof`

Property `TrapOnEof` has an integer type and defines device behaviour by default on detection end of stream to read. If this value is not 0, devices generate error. If value is 0, devices up system variable `$zeof`. By default for compatibility with legacy applications devices raises error. Device behaviour on detection end of read is depended of device type and applies only to devices for which this state can be defined, for example disk files or externally defined device in external DLL/ SO (ZDEVICE).

### FrameCount

Property `FrameCount` has an integer type and defines maximum level of stack frames of subroutines calls. This property is optional for specification and by default this value is equal to 1024. Minimum value is 16, Maximum is 131072.

### JournalCache

Property `JournalCache` has an integer type and defines size of cache of journal buffer in megabytes. This property is optional for specification, value by default is 8 megabytes. Minimum value is 1 megabyte, maximum for 32-bit architecture is 64 megabytes and for 64-bit architectures this value has no limits.

### ProcessStorage

Property `ProcessStorage` has an integer type and defines size of memory for local variables in megabytes. This property is optional and has default value of 8 megabytes. Minimum value is 1 megabyte, maximum for 32-bit architectures is 64 megabytes and for 64-bit architectures this property has no limits.

## 6.8.2 MiniMono.VM functions

### CreateVM

Function `CreateVM` creates instance of MiniMono virtual machine by initial settings specified by properties. Instance of MiniMono virtual machine can be created only one per process of operating system and only one instance per datafile.

Return values are: 0 if virtual machine was created successfully, 1 if instance of virtual machine already was created (in this or in other process for the same datafile) and 2 on other errors - not enough memory or internal operating system's objects.

Example:



```
Dim MiniMono
Set MiniMono = WScript.CreateObject( "MiniMono.VM")
```

```
MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0
```

```
MiniMono.CreateVM
```

```
MiniMono.FreeVM
Set MiniMono = Nothing
```

After function `CreateVM` any changes of values of properties are not affected and does not accepted.

#### FreeVM

Function `FreeVM` terminates instance of `MiniMono` virtual machine. Return value is absent. If `ActiveX` object does not contain active instance, this function does do anything.

Example:

```
Dim MiniMono
Set MiniMono = WScript.CreateObject( "MiniMono.VM")
```

```
MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0
```

```
MiniMono.CreateVM
```

```
MiniMono.FreeVM
Set MiniMono = Nothing
```

#### EvalStr

Function `EvalStr` calculates value of argument as a `MUMPS` expression in `MiniMono` dialect. Argument is a string (`BSTR` type) and return value is a string (`BSTR` type) with evaluation result.

Example:

```

Dim MiniMono
Set MiniMono = WScript.CreateObject( "MiniMono.VM")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

WScript.Echo MiniMono.EvalStr( "$zv")

MiniMono.FreeVM
Set MiniMono = Nothing

```

Eval

Function Eval calculates value of argument as a MUMPS expression in MiniMono dialect. Argument is an object of MiniMono.ServerString type and return value is an object of MiniMono.ServerString type with evaluation result.

Example:

```

Dim MiniMono
Set MiniMono = WScript.CreateObject( "MiniMono.VM")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

Dim Tmp
Set Tmp = WScript.CreateObject( "MiniMono.ServerString")
Tmp.Value = "$zv"

Dim Res
Set Res = MiniMono.Eval( Tmp)

WScript.Echo Res.Value

```

```
MiniMono.FreeVM  
Set MiniMono = Nothing
```

#### GetError

Function GetError return value of last occurred error and return type is an object of MiniMono.ServerString.

Example:

```
Dim MiniMono  
Set MiniMono = WScript.CreateObject( "MiniMono.VM")  
  
MiniMono.DataFile = "empty.dat"  
MiniMono.DBCacheSize = 100  
MiniMono.JournalingEnabled = 0  
  
MiniMono.CreateVM  
  
WScript.Echo "Undefined: " & MiniMono.EvalStr( "unknown")  
  
Dim ServerString  
Set ServerString = MiniMono.GetError  
  
WScript.Echo ServerString.Value  
  
MiniMono.FreeVM  
Set MiniMono = Nothing
```

#### GetErrorStr

Function GetErrorStr returns value of last occurred error and return type is a string (BSTR type).

Example:

```

Dim MiniMono
Set MiniMono = WScript.CreateObject( "MiniMono.VM")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

WScript.Echo "Undefined: " & MiniMono.EvalStr( "unknown")

WScript.Echo MiniMono.GetErrorStr()

MiniMono.FreeVM
Set MiniMono = Nothing

```

### Execute

Function `Execute` executes one line of commands defined by argument with type of `MiniMono.ServerString`. Return value is an integer number with value of error code:

0	Function executed successfully
1	Argument is not valid MUMPS commands because contains syntax errors
5	Error in database or in memory or other hardware or in operating system
6	Executed code did call the HALT command

Example:

```

Dim MiniMono
Set MiniMono = WScript.CreateObject( "MiniMono.VM")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

```

```
Dim Commands
Set Commands = WScript.CreateObject( "MiniMono.ServerString")

Commands.Value = "s list=$lb(123,456,$h)"

MiniMono.Execute Commands

Dim Tmp
Set Tmp = WScript.CreateObject( "MiniMono.ServerString")
Tmp.Value = "list"

Dim List
Set List = MiniMono.Eval( Tmp)

Dim Element
Set Element = WScript.CreateObject( "MiniMono.ServerString")

MiniMono.ListGet List, 1, Element
WScript.Echo "Before: " & Element.Value

Element.Value = "next"

MiniMono.ListSet List, 1, Element

MiniMono.ListGet List, 1, Element
WScript.Echo "After: " & Element.Value

MiniMono.FreeVM
Set MiniMono = Nothing

ExecuteStr
```

Function `ExecuteStr` executes one line of commands defined by argument with type of `BSTR`, ordinal string. Return value is an integer number with value of error code:

- |   |   |
|---|---|
| 0 | Function executed successfully                                      |
| 1 | Argument is not valid MUMPS commands because contains syntax errors |

- 5 Error in database or in memory or other hardware or in operating system
- 6 Executed code did call the HALT command

Example:

```
Dim MiniMono
Set MiniMono = WScript.CreateObject( "MiniMono.VM")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

MiniMono.ExecuteStr "s list=$lb(123,456,$h)"

Dim Tmp
Set Tmp = WScript.CreateObject( "MiniMono.ServerString")
Tmp.Value = "list"

Dim List
Set List = MiniMono.Eval( Tmp)

Dim Element
Set Element = WScript.CreateObject( "MiniMono.ServerString")

MiniMono.ListGet List, 1, Element
WScript.Echo "Before: " & Element.Value

Element.Value = "next"

MiniMono.ListSet List, 1, Element

MiniMono.ListGet List, 1, Element
WScript.Echo "After: " & Element.Value

MiniMono.FreeVM
Set MiniMono = Nothing
```

### SetTest

Function SetTest changes in runtime value of system variable \$TEST. Argument has type of integer number and, if was value of 0, system variable \$TEST changes to 0, otherwise changes to 1.

This function has not return value.

Function SetTest was intended to be applied in context of callback calls in device event handlers when value of \$TEST need to be changed by device call conventions, for example if reading was with timeout but timeout has expired.

Example:

```
MiniMono.SetTest 1
```

### SetCtrlBreak

Function SetCtrlBreak changes internal state of MiniMono virtual machine into interrupt context. MiniMono virtual machine on execution or expression evaluation periodically checks interruption state and, if interrupt indicator is on, virtual machine raises an error INTERRUPT.

Argument is an integer number and, if has value of 0, internal interruption state does not changes, otherwise changes.

Function SetCtrlBreak was intended to use in context of callback calls in device event handlers, or need to be called inside of handler of Ctrl+Break event, or when was pressed Ctrl+C or in other context which need to be interrupted.

This function can be called asynchronously too, not only inside of device event handlers.

Example:

```
MiniMono.SetCtrlBreak 1
```

### ListGet

Function ListGet returns one element of list structure.

First argument, with type of MiniMono.ServerString, must contains value of list structure, second argument, integer number, must define position of list item and third argument, reference to object of MiniMono.ServerString, accepts list item value.

This function has not return value.

If list item have undefined value, this function return empty string as a list item.

Example:

```
Dim MiniMono
Set MiniMono = WScript.CreateObject( "MiniMono.VM")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

MiniMono.ExecuteStr "s list=$lb(123,456,$h)"

Dim Tmp
Set Tmp = WScript.CreateObject( "MiniMono.ServerString")
Tmp.Value = "list"

Dim List
Set List = MiniMono.Eval( Tmp)

Dim Element
Set Element = WScript.CreateObject( "MiniMono.ServerString")

MiniMono.ListGet List, 1, Element
WScript.Echo Element.Value

MiniMono.ListGet List, 2, Element
WScript.Echo Element.Value

MiniMono.ListGet List, 3, Element
WScript.Echo Element.Value
```



```
MiniMono.FreeVM
Set MiniMono = Nothing
```

```
ListSet
```

Function ListSet changes item of list structure into specified value.

First argument, with type of MiniMono.ServerString, contains source list structure where list item need to be changed. Second argument, an integer number, defines position of list item. Third argument, object of MiniMono.ServerString, defines new list item value.

This function has not return value.

```
Dim MiniMono
Set MiniMono = WScript.CreateObject( "MiniMono.VM")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

MiniMono.ExecuteStr "s list=$lb(123,456,$h)"

Dim Tmp
Set Tmp = WScript.CreateObject( "MiniMono.ServerString")
Tmp.Value = "list"

Dim List
Set List = MiniMono.Eval( Tmp)

Dim Element
Set Element = WScript.CreateObject( "MiniMono.ServerString")

MiniMono.ListGet List, 1, Element
WScript.Echo "Before: " & Element.Value

Element.Value = "next"
```

```

MiniMono.ListSet List, 1, Element

MiniMono.ListGet List, 1, Element
WScript.Echo "After: " & Element.Value

MiniMono.FreeVM
Set MiniMono = Nothing

```

### ListLength

Function ListLength returns number of items in list structure including undefined items.

Argument has type of MiniMono.ServerString and defines list structure.

Example:

```

Dim MiniMono
Set MiniMono = WScript.CreateObject( "MiniMono.VM")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

MiniMono.ExecuteStr "s list=$lb(123,456,$h)"

Dim Tmp
Set Tmp = WScript.CreateObject( "MiniMono.ServerString")
Tmp.Value = "list"

Dim List
Set List = MiniMono.Eval( Tmp)

WScript.Echo MiniMono.ListLength( List)

MiniMono.FreeVM
Set MiniMono = Nothing

```

## Text

Function Text decorates source string by MUMPS language syntax conventions. First argument, with type of MiniMono.ServerString, defines source string as a byte sequence. Second argument defines object of MiniMono.ServerString, which must accept result of decoration.

Function Text executes text decoration as such as function \$ZQUOTE to create correct string in MUMPS syntax for strings with nonprintable bytes. Function substitutes instead of nonprintable bytes concatenation operator and \$CHAR functions, and doubles, if need, double quotes. Function counts bytes as unsigned characters.

Example:

```
Dim MiniMono
Set MiniMono = WScript.CreateObject( "MiniMono.VM")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

MiniMono.ExecuteStr "s list=$lb(123,456,$h)"

Dim Tmp
Set Tmp = WScript.CreateObject( "MiniMono.ServerString")
Tmp.Value = "list"

Dim Res
Set Res = MiniMono.Eval( Tmp)

WScript.Echo Res.Value

Dim Str
Set Str = WScript.CreateObject( "MiniMono.ServerString")

MiniMono.Text Res, Str

WScript.Echo Str.Value
```

```
MiniMono.FreeVM
Set MiniMono = Nothing
```

TextStr

Function Text decorates source string by MUMPS language syntax conventions. Argument, with type of ordinal string (BSTR type), defines source string as a byte sequence. Function returns ordinal string (BSTR type) with the result of decoration.

Function TextStr executes text decoration as such as function \$ZQUOTE to create correct string in MUMPS syntax for strings with nonprintable bytes. Function substitutes instead of nonprintable bytes concatenation operator and \$CHAR functions, and doubles, if need, double quotes. Function counts bytes as unsigned characters.

Example:

```
Dim MiniMono
Set MiniMono = WScript.CreateObject( "MiniMono.VM")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

WScript.Echo MiniMono.TextStr( "str"str")

MiniMono.FreeVM
Set MiniMono = Nothing
```

### 6.8.3 MiniMono.VM events

Events of object MiniMono.VM are called from inside of context of MiniMono virtual machine when appropriate event occurs. In other word, events are fired when MUMPS application need to execute some operations with principal device. Event handlers can be assigned independently from each other, and if one or mor eevent handlers was not assigned, MiniMono virtual machine uses default behaviour for this operation.

All event handlers has return value as an integer number, and must return 0 on success or other value on error. If execution environment of ActiveX object does not supports return value of event handlers, MiniMono virtual machine counts this as success execution by default.

`DevUse(PairsCount,ParamPairs)`

Event handler `DevUse` calls by MiniMono virtual machine on execution of the `USE` command for principal device.

First argument is an integer number and defines number of pairs of key and values was passed to the `USE` command as options of command. Second argument is an array of values in the order of key of option + value of option. This array is passed as array of `VARIANT`, one item of array is object of `VARIANT`. If name of option or value of option was omitted, there are passed an empty string.

Example:

```
Dim MiniMono
Set MiniMono =
    WScript.CreateObject( "MiniMono.VM", "MiniMono_" )

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

Dim Val
Dim Expr

Set Expr =
    WScript.CreateObject( "MiniMono.ServerString" )
Expr.Value = "$zv"

Set Val = MiniMono.Eval( Expr )

WScript.Echo Val.Value

MiniMono.ExecuteStr "u $p: (/P1=123:/P2=456)"
```

```

MiniMono.FreeVM

Set Expr = Nothing
Set MiniMono = Nothing

Sub MiniMono_DevUse( PairsCount, Pairs)
  For Counter = 0 to PairsCount - 1
    ParamName = Pairs( Counter * 2)
    ParamValue = Pairs( Counter * 2 + 1)
    WScript.Echo
      "DevUse, Parameter = " & ParamName &
      " Value = " & ParamValue
  Next
End Sub

```

```
DevWriteStr(Value)
```

Event handler `DevWriteStr` calls when MUMPS application need to execute the `WRITE` command to principal device.

Argument of event handler is an object of type `MiniMono.ServerString` and contains set of bytes need to be written to the principal device. Write method in real must be defined by developer.

Example:

```

Dim MiniMono
Set MiniMono =
  WScript.CreateObject( "MiniMono.VM", "MiniMono_")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

MiniMono.ExecuteStr "w 123"

MiniMono.FreeVM

```

```

Set MiniMono = Nothing

Sub MiniMono_DevWriteStr( Str)
    WScript.Echo
    "Event DevWriteStr fired, str = " &
    str.Value
End Sub

```

```
DevWriteChar(Value)
```

Event handler `DevWriteChar` executes when MUMPS application executes the `WRITE` command with symbol code.

Argument of event handler have a type of an integer number and here can be passes negative as such as positive codes. Method of writing this code must be defined by developer. In most cases writing og code means writing one byte with this code, but in some cases developers can define special behaviour for specially reserved codes (mainly negative values), and replace by such way, the `USE` command.

Example:

```

Dim MiniMono
Set MiniMono =
    WScript.CreateObject( "MiniMono.VM", "MiniMono_")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

MiniMono.ExecuteStr "w *122,*123"

MiniMono.FreeVM

Set MiniMono = Nothing

Sub MiniMono_DevWriteChar( Symbol)

```

```

    WScript.Echo
      "Event DevWriteChar fired, Symbol = " & Symbol
End Sub

```

DevWriteNL()

Event handler DevWriteNL executes on execution formatted output of line feed and has not arguments.

Example:

```

Dim MiniMono
Set MiniMono =
  WScript.CreateObject( "MiniMono.VM", "MiniMono_")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

MiniMono.ExecuteStr "w !!"

MiniMono.FreeVM

Set MiniMono = Nothing

Sub MiniMono_DevWriteNL
  WScript.Echo "Event DevWriteNL fired"
End Sub

```

DevWriteFF()

Event handler DevWriteFF executes on execution of formatted writing of form feed (new page) and has not arguments.

Example:



```

Dim MiniMono
Set MiniMono =
    WScript.CreateObject( "MiniMono.VM", "MiniMono_")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

MiniMono.ExecuteStr "w ##"

MiniMono.FreeVM

Set MiniMono = Nothing

Sub MiniMono_DevWriteFF
    WScript.Echo "Event DevWriteFF fired"
End Sub

DevWriteTAB(TabCount)

```

Event handler DevWriteTAB executes when MUMPS code executes formatted writing with tabulation to the principal device. Argument TabCount has an integer type and contains value passed from the MUMPS code in the tabulation argument of the WRITE command.

Example:

```

Dim MiniMono
Set MiniMono =
    WScript.CreateObject( "MiniMono.VM", "MiniMono_")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

MiniMono.ExecuteStr "w ?5,?12"

```

```

MiniMono.FreeVM

Set MiniMono = Nothing

Sub MiniMono_DevWriteTAB( TabCount)
  WScript.Echo
  "Event DevWriteTAB fired, TabCount = " &
  TabCount
End Sub

DevReadStr(Len,TimeOut,ReadString)

```

Event handler DevReadStr executes when the MUMPS code executes the READ command for the principal device.

Argument Len has an integer type and contains how many bytes was specified in the READ arguments for reading. If command READ was used without specification of length for reading, this argument contains value of -1. Maximum length for reading allowed is dependent of internal MiniM architecture and is equal to 32 kilobytes.

Argument TimeOut has an integer type and contains number of milliseconds for read timeout. Command READ contains option for timeout in seconds, but MiniMono virtual machine recalculates this value to milliseconds, so command READ can contain fractional values. If command READ was used without timeout specification, this argument contains value of -1.

Argument ReadString has type of MiniMono.ServerString and event handler must place into this object read result as a byte sequence. This byte sequence will be used as a result of the READ command.

Example:

```

Dim MiniMono
Set MiniMono =
  WScript.CreateObject( "MiniMono.VM", "MiniMono_")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

```

```

MiniMono.CreateVM

MiniMono.ExecuteStr "r str"

WScript.Echo "str after read is " &
  MiniMono.EvalStr( "str")

MiniMono.FreeVM

Set MiniMono = Nothing

Sub MiniMono_DevReadStr( Length, TimeOut, Str)
  WScript.Echo "Event DevReadStr fired"
  Str.Value = "answer"
End Sub

DevReadChar(TimeOut,ReadChar)

```

Event handler `DevReadChar` executes when MUMPS code execute the `READ` command for one character (number of character code) from the principal device.

Argument `TimeOut` has an integer number type and defines timeout in milliseconds. Command `READ` contains option for timeout in seconds, but `MiniMono` virtual machine recalculates this value to milliseconds, so command `READ` can contain fractional values. If command `READ` was used without timeout specification, this argument contains value of -1.

Argument `ReadChar` has type of `VARIANT`. Event handler should assign to this argument integer number of code has been read. `MiniMono` virtual machine counts numbers of code as unsigned characters (bytes) with code from 0 to 255.

Example:

```

Dim MiniMono
Set MiniMono =
  WScript.CreateObject( "MiniMono.VM", "MiniMono_")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100

```

```

MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

MiniMono.ExecuteStr "r *ch"

WScript.Echo
    "ch after read is " & MiniMono.EvalStr( "ch")

MiniMono.FreeVM

Set MiniMono = Nothing

Sub MiniMono_DevReadChar( TimeOut, ReadCode)
    WScript.Echo "Event DevReadChar fired"
    ReadCode = 123
End Sub

DevGetX(Value)
DevGetY(Value)

```

Event handlers DevGetX and DevGetY executes when MUMPS code execute reading of values of system variables \$X and \$Y for the principal device.

Argument Value has type of VARIANT and event handler should assign to this argument values of \$X or \$Y.

What does mean values of system variables \$X and \$Y of the principal device in current application developer should define self.

Example:

```

Dim MiniMono
Set MiniMono =
    WScript.CreateObject( "MiniMono.VM", "MiniMono_")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

```

```

MiniMono.CreateVM

MiniMono.ExecuteStr "s x=$x,y=$y"

WScript.Echo "values are: x = " &
  MiniMono.EvalStr( "x") &
  ", y = " & MiniMono.EvalStr( "y")

MiniMono.FreeVM

Set MiniMono = Nothing

Sub MiniMono_DevGetX( XValue)
  WScript.Echo "Event DevGetX fired"
  XValue = 123
End Sub

Sub MiniMono_DevGetY( YValue)
  WScript.Echo "Event DevGetY fired"
  YValue = 456
End Sub

DevSetX(Value)
DevSetY(Value)

```

Event handlers DevSetX and DevSetY executes when MUMPS code execute assignment to system variables \$X and \$Y for principal device.

Argument Value has type integer number and here is passed value assigned in the MUMPS code. If in MUMPS code value was not true integer, MiniMono virtual machine casts this value to integer by MUMPS casting convention.

What does mean values of system variables \$X and \$Y of the principal device in current application developer should define self.

Example:

```

Dim MiniMono
Set MiniMono =
  WScript.CreateObject( "MiniMono.VM", "MiniMono_")

```

```

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

MiniMono.ExecuteStr "s $x=123,$y=456"

MiniMono.FreeVM

Set MiniMono = Nothing

Sub MiniMono_DevSetX( XValue)
    WScript.Echo
        "Event DevSetX fired, XValue = " & XValue
End Sub

Sub MiniMono_DevSetY( YValue)
    WScript.Echo
        "Event DevSetY fired, YValue = " & YValue
End Sub

DevGetKEY(KeyValue)

```

Event handler DevGetKEY executes when MUMPS code reads value of system variable \$KEY for principal device.

Argument KeyValue has type of MiniMono.ServerString and event handler should assign to his property Value byte sequence correspondent to the \$KEY for principal device. Developer of event handler should define self what does mean value of \$KEY for principal device. In most cases this variable contains last read terminator.

Example:

```

Dim MiniMono
Set MiniMono =
    WScript.CreateObject( "MiniMono.VM", "MiniMono_")

```

```

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

MiniMono.ExecuteStr "s key=$key"

WScript.Echo "key has value " &
  MiniMono.EvalStr( "key")

MiniMono.FreeVM

Set MiniMono = Nothing

Sub MiniMono_DevGetKEY( Key)
  WScript.Echo "Event DevGetKEY fired"
  Key.Value = "abcd"
End Sub

DevSetKEY(KeyValue)

```

Event handler DevSetKEY executes when MUMPS code execute assigning of system variable \$KEY for the principal device.

Argument KeyValue has type of MiniMono.ServerString and his property Value have a byte sequence of value passed from the MUMPS code on assigning.

Developer should decide whate mean assigning of system variable \$KEY for principal device in developed application.

Example:

```

Dim MiniMono
Set MiniMono =
  WScript.CreateObject( "MiniMono.VM", "MiniMono_")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

```

```

MiniMono.CreateVM

MiniMono.ExecuteStr "s $key=123"

MiniMono.FreeVM

Set MiniMono = Nothing

Sub MiniMono_DevSetKEY( Key)
    WScript.Echo
    "Event DevSetKEY fired: " & Key.Value
End Sub

DevZEOF(Value)

```

Event handler DevZEOF executes when MUMPS code execute reading value of \$ZEOF for principal device.

Argument Value has type VARIANT and in this variable should be written indicator 0 if reading did reached end of input byte sequence or not 0 if not.

All event handlers are in callback calling state, so they cannot create true error raising as such as internal builtin devices when reach end of read state. In common words, \$ZEOF usage is much more recommended way to handle end-of-file state of developed device.

Example:

```

Dim MiniMono
Set MiniMono =
    WScript.CreateObject( "MiniMono.VM", "MiniMono_")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

MiniMono.ExecuteStr "s zeof=$zeof"

```



```

WScript.Echo "zeof value is: " &
  MiniMono.EvalStr( "zeof")

MiniMono.FreeVM

Set MiniMono = Nothing

Sub MiniMono_DevZEOF( ZEOF)
  WScript.Echo "Event DevZEOF fired"
  ZEOF = 1
End Sub

DevGetZA(KeyValue)
DevGetZB(KeyValue)

```

Event handlers DevGetZA and DevGetZB executes when MUMPS code execute reading values of system variables \$ZA and \$ZB for principal device.

Argument KeyValue has type of MiniMono.ServerString and his property Value must be assigned to byte sequence of variables \$ZA and \$ZB, and this sequences will get MUMPS code.

What does mean values of system variables \$ZA and \$ZB must define developer of principal device. Traditionally this variables are used for backward compatibility with legacy code and other MUMPS systems to get some device-specific states.

Example:

```

Dim MiniMono
Set MiniMono =
  WScript.CreateObject( "MiniMono.VM", "MiniMono_")

MiniMono.DataFile = "empty.dat"
MiniMono.DBCacheSize = 100
MiniMono.JournalingEnabled = 0

MiniMono.CreateVM

MiniMono.ExecuteStr "s za=$za,zb=$zb"

```

```

WScript.Echo "za value is: " &
  MiniMono.EvalStr( "za")
WScript.Echo "zb value is: " &
  MiniMono.EvalStr( "zb")

MiniMono.FreeVM

Set MiniMono = Nothing

Sub MiniMono_DevGetZA( ZA)
  WScript.Echo "Event DevGetZA fired"
  ZA.Value = 123
End Sub

Sub MiniMono_DevGetZB( ZB)
  WScript.Echo "Event DevGetZB fired"
  ZB.Value = 456
End Sub

```

#### 6.8.4 MiniMono.ServerString properties

Value

*Value* is a property of *MiniMono.ServerString* object with type of simple string. This property is accessible for read and write.

Length

*Length* is a property of *MiniMono.ServerString* object with type of Integer and contains real bytes count been used in object. This property is accessible for read and write. When assigned to value less than 0 property sets to value 0, when assigned to value greater than 32767, property sets to value 32767.

#### 6.8.5 MiniMono.ServerString functions

GetAt( Pos)

Function returns integer as character code at position *Pos*, where position is specified by integer. If *Pos* show outside of bytes are available, function returns code -1. Character code is returned as integer from 0 to 255.

**SetAt( Pos, Code)**

Function replaces byte at position *Pos* to byte with code *Code*. Arguments *Code* and *Pos* must be an integers. If value of *Pos* shows outside of bytes are available, function does nothing.

**Add( Val )**

Function concatenates to internal object data data of other object *Val* of type *MiniMono.ServerString*. Value of *Val* does not changes. If function executes successfully, function return value 1, otherwise value 0.

**AddStr( Val)**

Function concatenates to internal object data bytes of string *Val*. Value of *Val* does not changes. If function executes successfully, function return value 1, otherwise value 0.



# Chapter 7

## MiniMono for Android

### 7.1 SDK Content

MiniMono SDK for Android consists of set of files need to be used for application development for Android with embedding of MiniMono Embedded Edition.

MiniMono SDK does not has installer and uninstaller, does not contains any utilities need to be run on development computer and does not depends of operating system on the development computer. To use the MiniMono SDK for application development this file set is used with Android SDK and any development tools for Android applications. Id developer need to edit any MiniMono - specific files, there must be used full MiniMono installation for development operating system. This custom files may be datafiles with need set of routines, or routines, or globals, or collation files, or any others. And, of course, can be used full MiniM Database Server as a developer tool for MUMPS development.

MiniMono SDK distributes as a ZIP archive and after unpacking developer have the following subdirectories:

db	Datafiles for MiniM, and by default this subdirectory contains at least one ampty datafile empty.dat
doc	MiniM documentation in the PDF format
examples	Examples for Android in the Java language to use with the Android SDK
include	Header and interface files for low-level development and integration of modules in C or Pascal (FreePascal + Lazarus)

libs	Compiled libraries libminimono.so + libminimonoj.so + libzini.so for different target Android architectures and processors
routines	Standard set of system routines for MiniM
src	minimonoj and utils modules in the Java language to develop applications with the Android SDK
zdevice	Examples of development external ZDEVICE modules in C and Pascal
zdll	Examples of development external ZDLL modules in C and Pascal

At the current time MiniMono SDK supports the following processor architectures for Android:

armeabi	Set of processors based on the ARM architecture, 32-bit.
x86	Set of processors based on the Intel/AMD x86-32 architecture, from the i386 and later, 32-bit.
arm64-v8a	Set of processors based on the ARM, 64-bit.
x86_64	Set of processors based on the Intel/AMD x86-64, 64-bit.

At the current time mostly wide used are devices on Android based on the ARM and x86 processors, and 64-bit processors are rarely used. In most cases to develop application for Android including processor architectures of armeabi and x86 are enough..

At the current time MiniMono SDK does not supports processor architectures of MIPS and MIPS64, because this processors use big-endian byte order and MiniM does not ported to any of big-endian processor architecture, such as MIPS, Itanium, SPARC, etc.

File libminimono.so contains module of MiniMono and is main executable for MiniMono virtual machine. This file have full functionality of the other MiniMono versions and fully compatible with MiniM by MUMPS language dialect. This file is at the minimum need to embed MiniMono virtual machine into Android application.

File libminimonoj.so is a JNI interface between Java and MiniMono. This file is need to build application for Android using Java. Examples for MiniMono SDK contains Java code and are use this module. If developer uses

other language to develop Android application (C or Pascal or others), this file is unneeded.

File `libzini.so` is an external ZDLL module with external functions for routine `%INI`. This routine is present in the standard MiniM set of system routines, but default empty datafile `empty.dat` does not contain this routines. If developers does not use the `%INI` routine, this file is unneeded. Examples for MiniMono SDK does not use this file.

## 7.2 Building Application

Applying of MiniMono SDK for Android was illustrated with Android SDK from Google and main developer's guide is on the site:

<http://developer.android.com>

On this site can be found any need tools, information for Android development, building rules and application distribution rules.

Building of application for Android are illustrated on the Java language as main execution context. If developers are used any other development tools for application development for Android, only file `libminimono.so` and header and interface file need to be used.

To create application need to include the tools subdirectory from the Android SDK from Google into the search path (the `PATH` environment variable). And, to create initial subdirectory set of application, need to be used utility:

```
android create project
```

And here can be recommended to create batch file, with can automate this operations, for example, to create application `Example1` on Windows x64 this file can be the followed:

```
set PATH=C:\Program Files (x86)\Android\  
    android-sdk\tools;%PATH%  
md example1  
android create project --target 1  
    --name example1 --path example1  
    --activity Example1  
    --package android.database.minimono
```

Here we create subdirectory `example1` and inside of one this tool automatically creates need subdirectories by default for `example1` application, with main Activity `Example` in the Java package

```
android.database.minimono
```

After creation of initial application skeleton we must add to this project library files of `libminimono.so` and `libminimonoj.so` from the appropriate `libs` subdirectory of the MiniMono SDK into the `libs` subdirectory of application to make the following directory tree:

```
/example1
  /libs
    /armeabi
      libminimono.so
      libminimonoj.so
    /x86
      libminimono.so
      libminimonoj.so
```

This `so` files from the MiniMono SDK are used in the `/libs` together with other `so` files from project, if they are present, And, in this subdirectories we must place external `ZDLL` and `ZDEVICE` `so` modules if they are used in the application.

On the build of application (to get the `apk` file) this libraries will be inserted into the installer and on the target Android device will be used only files with appropriate processor architecture.

In the `/src` subdirectory of the project we must place subdirectory `/android`, inside of one subdirectory `/database`, inside of one subdirectory `/minimono`, and inside of one the following files: `minimonoj.java` and `utils.java`. So, we must get the following directory tree:

```
/src
  /android
    /database
      /minimono
        minimonoj.java
        utils.java
```



This subdirectory structures was defined by conventions of the Java language. And, in this subdirectories, we must place also files of examples ExampleXXX.java to get full source file set for example.

Next, in the file /res/layout/main.xml need to create application interface structure. In simple case this file can be replaced by the file main.xml from the MiniMono SDK.

For proper MiniMono working this virtual machine requires data file. MiniMono SDK contains subdirectory /db with empty datafile empty.dat. All examples uses file transfer of this datafile thought the /assets/db subdirectory of project. All files was placed into /assets and subdirectories of /assets, apk creation tool embeds as is, in raw state. This files are accessible inside of Android application throught the AssetManager object as pseudofiles. Utilities from the file

```
/android/database/minimono/utils.java
```

are used by examples for file synchronization from the Assets storage into the local file system on Android, by conventions of Android. This files after extracting are raw and true files in local filesystem and can be used as datafiles and other files by MiniMono.

After preparing directory structure for Android application this project is ready for building. Android SDK recommends to use the ANT build system. This system requires that the own /bin directory and /bin directory of the JDK was in the search path (the PATH environment variable). So, may be recommended create batch file with commands to automate this actions. Windows version of this batch file can be the followed:

```
set ANT_HOME=c:\tools\apache-ant
set PATH=%ANT_HOME%\bin;%PATH%
set JAVA_HOME=C:\Program Files\Java\jdk1.8.0_20
set PATH=%JAVA_HOME%\bin;%PATH%
cd example1
ant debug
```

In the case of usage JDK of other versions or other directories here need to be used real directory names.

After building project in the /bin subdirectory of project's directory tree will be final apk file, for example, example1-debug.apk.

To build release apk version need to be used appropriate utilities from Google and build rules from Google to subscribe distribution file.

To place and install apk file on the Android device need to be used adb utility with command line parameter

```
adb install
```

and to reinstall application with the parameters

```
adb install -r
```

with specifying what device need to be used and what apk file must be installed.

All MiniMono examples uses very simple user interface, was made in the Java language and build steps are the same.

To supply not only 32-bit processors arm and x86, but 64-bit processors too, there need to include appropriate /libs subdirectories arm64-v8a and x86\_64 with libminimono.so and libminimonoj.so libraries.

All examples does not use external collation definition files nat, all examples use character collation rules by default. To use appropriate character collation rule need to transfer this nat file too through the /assets storage, for example in the /nat subdirectory, and after synchronisation specify real local file name in MiniMono initialisation.

### 7.3 Assets Synchronisation Utility

Assets - this is additional structured file storage for Android application, with content which does not transform by apk builder in any case. This files with all subdirectory structure transfers as is, in raw mode. This file storage is used and recommended to transfer all additional files for Android application.

To use in Android application module of MiniMono virtual machine developer must transfer at least one datafile. Optionally, by application developer decision, can be transferred nat files, source code of routines in separate files to import later into MiniMono context, or other files.

Instead of empty datafile, as this was done in examples, developers can transfer special prepared datafile for MiniMono with need set of compiled routines and initial global's state.

Before transferring raw files in the /assets storage in the apk file developers must understand that Android developers can implement his own limits for this files by size. This limits can be applied by Google developers. In the case of this limit was reached, application developer must split file content to several files with smaller size and unite this data in the working application on Android or use other source of data (for example, download from the internet).

All files was transferred in the Assets storage, are accessible as special pseudofiles, throught the AssetManager object. To place files from the assets to local file system, to use by the MiniMono, this files must be copied from the storage to local file system. All this actions do special utility class utils.

Synchronisation utilities for assets reads list of available in the storage files and for each file checks existence of appropriate file in local file system If this file exists, this file does not touch, otherwise this file creates and copies from the assets storage as is, in initial state.

application developers for Android, of course, can use any other synchronisation utilities for assets and any own application rules to synchronise files.

Before usage of assets synchronisation utilities file utils.java must be placed in the project directory before build apk.

Function of class utils

```
public void syncAssetDir( String dir,
    Context context, AssetManager assets)
```

executes synchronisation all of available files in the assets storage specified in the dir subdirectory of assets.

Function of class utils

```
public String fileName( String dir,
    Context context, String name)
```

returns real file name in the local file system correspondent to specified file name and subdirectory of assets storage.

## 7.4 Examples

Examples for MiniMono for Android was made as applications in the Java language with using Java object with JNI module (libminimonoj.so), and this module calls to MiniMono virtual machine in the libminimono.so library:

```

import android.database.minimono.minimonoj;
import android.database.minimono.utils;

...

minimonoj MiniMono =
    new minimonoj( CodePage);
minimonoj.MiniMonoVM init =
    new minimonoj.MiniMonoVM( CodePage);
int ret = MiniMono.CreateMiniMono( init);

```

The screen of application consists of two controls - textbox and button. Textbox displays outputted text, and button click handler terminates application.

All examples use the same initialisation of MiniMono virtual machine:

```

MiniMono.GetDefaultSettings( init);
init.DataFile =
    u.fileName( dbDir,
        getApplicationContext(), dbFileName);
init.JournalingEnabled = 0;

```

Before initialisation applications do synchronisation of datafiles from the assets storage:

```

utils u = new utils();
String dbDir = "db";
String dbFileName = "empty.dat";
u.syncAssetDir( dbDir,
    getApplicationContext(), getAssets());

```

And virtual machine of MiniMono uses real local file name in the local file system after synchronisation:

```

init.DataFile =
    u.fileName( dbDir,
        getApplicationContext(), dbFileName);

```

Next, examples executes sequence of calls to MiniMono virtual machine and next terminates virtual machine by

```
MiniMono.FreeMiniMono();
```

Total application execution terminates by clicking to the button:

```
final Button btnClose =
    (Button)findViewById(R.id.btnClose);
btnClose.setOnClickListener(
    new View.OnClickListener() {
        public void onClick(View v) {
            finish();
        }
    });
```

Example Example1.java demonstrates evaluation of the MUMPS expression and reading of result. Result outputs to the textbox on the application's screen:

```
txtOut.append( "Value of $zversion: " +
    MiniMono.Read( "$zversion") + "\n");
txtOut.append( "Now is: " +
    MiniMono.Read( "$zdate($h,3)") + "\n");
```

Example Example2.java demonstrates execution of the set of MUMPS commands where assign values to local variables and after this reads back values of this local variables:

```
int n = 5;
MiniMono.Execute( "f i=1:1:" +
    n + " s a(i)=i*i");

for( int i = 1; i <= n; i++)
{
    txtOut.append(
        "Value of a(" + i + ") is: " +
        MiniMono.Read( "a(" + i + ")") +
        "\n");
}
```

Example `Example3.java` demonstrates working with variables with list structures. Example generates inside of MiniMono virtual machine value with list structure and after this reads this value and calls to list functions of MiniMono to display list length and each of list items:

```
MiniMono.Execute(
    "s var=$lb($zv,$sy,$zd($h,8))");
String list = MiniMono.Read( "var");

int n = MiniMono.ListLength( list);
txtOut.append(
    "Actual length of the list is: " +
    n + "\n");

for( int i = 1; i <= n; i ++)
{
    txtOut.append( "Item " + i + " : " +
        MiniMono.ListGet( list, i) + "\n");
}
```

Nest this examples shows how on the Java side change values of Java variables with list structures:

```
list = "";
// set 1 list item
list = MiniMono.ListSet(
    list, 1, "123456");
// set 2 list item
list = MiniMono.ListSet(
    list, 2, "123.456");
// set 3 list item
list = MiniMono.ListSet(
    list, 3, "Hello");
```

By the Java language conventions code cannot change value of the String object passing by reference or pointer, so functions returns entire list value to reassign entire value of the Java local variable.

Next this examples demonstrates usage of string decoration function by MUMPS syntax conventions. Function `MiniMono.Text` returns string with MUMPS syntax conventions. Function transforms source byte sequence into

string sequence with adding if this need concatenation operator, \$CHAR function and additional double quotes by needs.

Example Example4.java demonstrates how to define in Java behaviour of the principal device. To define this need to derive class `minimonoj` and overload appropriate set of virtual functions for device - how to write string, how to write one symbol with given code, line feed and other actions, with need to be used in Android application.

```
class minimonodev4 extends minimonoj
{
    private TextView txtOut;
    minimonodev4( String _CodePage,
        TextView _txtOut)
        throws java.lang.Exception
    {
        super( _CodePage);
        txtOut = _txtOut;
    };

    // overload functions for device handlers
    public int DevWriteStr( String str)
    {
        txtOut.append( "Fired WriteStr(\"" +
            str + "\") event\n");
        return 0;
    };
    public int DevWriteChar( int _Char)
    {
        txtOut.append( "Fired WriteChar(" +
            _Char + ") event\n");
        return 0;
    };
    public int DevWriteNL()
    {
        txtOut.append( "Fired WriteNL event\n");
        return 0;
    };
    ...
};
```

Next this derived class with overloaded event handlers of device is used in the application:

```
minimonodev4 MiniMono =
    new minimonodev4( CodePage, txtOut);
```

Next this example executes commands in MUMPS with the WRITE command to the principal device:

```
MiniMono.Execute(
    "write \"string\",*46,?12,!,#");
```

Example Example5.java demonstrates calls to indexed variables on short example how to read and write global's values. Before this calls example generates initial values of this global:

```
MiniMono.Execute(
    "f i=0:1:" + limit + " s ^var(i,i*i)=i*i*i");
```

To pass global and indices this example passes global name and array of indices:

```
// read global values using direct call
String[] Indices = new String[ 2];

for( int i = 0; i < limit; i++)
{
    Indices[ 0] = Integer.toString( i);
    Indices[ 1] = Integer.toString( i * i);
    txtOut.append(
        MiniMono.ReadGlobal( "var", Indices) + "\n");
}
```

To define name without any indices functions pass empty array of indices:

```
MiniMono.KillGlobal( "var", null);
```



This way is used by many other functions for local and global variables - ReadLocal, WriteLocal, KillLocal, OrderLocal, and others, and for passing separate function and subroutine arguments by UserFunc UserDo.

Object of array of strings in the Java language contains information about length of array and defines order of elements. In the case of absence any indices or arguments must be passed empty object null.

Example Example6.java demonstrates writing to the database source code of routine

```
// kill routine before create new
String global_name = "ROUTINE";
String routine_name = "Example6";
String[] indices = new String[ 1];
indices[ 0] = routine_name;
MiniMono.KillGlobal( global_name, indices);

// create routine as a series of direct global sets
indices = new String[ 2];
indices[ 0] = routine_name;

String[] routine_text =
{
    "Func(arg)",
    " w \"arg = \",arg,!\"",
    " n expr=$zd($h,8)",
    " q expr"
};
for( int i = 0; i < routine_text.length; i++)
{
    indices[ 1] = Integer.toString( i + 1);
    MiniMono.WriteGlobal(
        global_name, indices, routine_text[ i]);
}
```

After this done, example executes routine compilation into bytecode

```
// compile routine
MiniMono.Read(
    "$v(\"rou\", \"c\", \"\" + routine_name + "\")");
```

And after this calls compiled routine, executes mumps code inside of MiniMono virtual machine:

```
// evaluate expression
String func_result =
    MiniMono.Read( "$$Func^" +
        routine_name + "("any data\");
```

MUMPS code execution in this example demonstrates by output to the screen while executes the \$\$Func function and also by output to textbox result of this function.

Example Example7.java demonstrates error handling. JNI object for MiniMono on error occurrence inside of virtual machine generates native Java exception. This exception caught by the application code and application outputs on the screen special diagnostic message. This example uses two error cases - division by zero and reading undefined local variable.

```
try
{
    // execute division by zero
    MiniMono.Execute( "w 1/0");
}
catch( Throwable e)
{
    // catch execution error and display last error
    txtOut.append( "Caught an error: " +
        MiniMono.GetLastError() + "\n");
    // revert value of $ec to an empty
    // string for next code
    MiniMono.Execute( "s $ec=\"\");
}

try
{
    // read value of undefined variable
    MiniMono.Read( "abcdef");
}
catch( Throwable e)
{
    // catch execution error and display last error
```

```
txtOut.append( "Caught an error: " +
    MiniMono.GetLastError() + "\n");
// revert value of $ec to an empty
// string for next code
MiniMono.Execute( "s $ec=\"\"");
}
```

Here both exception handlers except output text to the screen of diagnostic message also clears value of the system variable \$ECODE (assign to the empty string). This can be used to clear error context of virtual machine for next calls. This \$ECODE variable clearing is unneed for proper use of MiniMono, but some error handlers in MUMPS can analyze full value of \$ECODE and can get redundant data about error which does not fully relate to error in MUMPS. Value of function MiniMono.GetLastError is simply value of the \$ZERROR system variable, this value does not accumulates as \$ECODE and displays information only about last accured error. Developers of application for Android with embedding MiniMono virtual machine should define own error handling by own conventions and coding style.