MiniM Database Server
Language Guide

Version 1.28
Eugene Karataev
mailto:support@minimdb.com
http://www.minimdb.com

April 15, 2017

Contents

Chapter 1

Syntax

1.1 Overall review

MiniM Database Server conforms to current MUMPS standard and hold MDC recommendations.

MiniM Database Server is a multiprocess server with caching datafile pages and routine's bytecodes.

Executable part of MiniM is an interpreter of compiler type. Interpreter executes compiled bytecode. Bytecode is created once on routine compilation. Interpreter use temporary bytecode too even for *xecute* command and for all indirection forms.

MiniM Database Server use extended syntax in the manner of MUMPS standard for extensions provided for MUMPS vendors.

MiniM supports transactions and data fault protection using combined journaling and before image writing. MiniM protect not datafile, but logical data in total of datafile, journal and before image journal.

In addition to standard MUMPS commands and functions MiniM Database Server implement several extensions such as:

- *new* command with initialization
- *$qsubscript()* function with assignment
- hexadecimal numbers
- read and kill from some structured system variables
- lazy logic operators

- list functions
- transactional bit functions
- external functions in dll
- external device types in dll
- and others

Also MiniM Database Server supports compatibility with some other MUMPS systems in syntax and in data structures and formats to implement extended application portability.

1.2 Commands

All actions been made by MiniM processes, are commands execution. Common command structure is a command keyword (what action need to perform), after command can be optional postconditional expression delimited by colon and optional argument delimited by space. Command can be applied to one or more arguments delimited by comma. Commads applies to arguments in left-to-right order. Only 3 commands cannot be specified with postconditional expressions - *for*, *if* and *else* because this commands are applied to all following commands in line.

Commands keywords are cese insensitive and can be abbreviated, in most cases to one symbol.

Command can be with argument or without argument, in second case this form is argumentless form of command and command applies not to specified argument, but to common process context. For example, the *kill* command with argument remove only specified variable but the *kill* command without arguments remove all available local variables. Or, for example, command *lock* with argument create locking of specified variable, but command *lock* without arguments removes all avalable locks performed by current process.

Unlike of most other programming languages, MUMPS command is a minimal possible part of execution.

Full list of commands supported by MiniM Database Server is described in current document.

One line of code can contain zero, one or more commands followed one after another and execution control follows by command sequence in left-to-right order one by one.

It is not supported execute only argument or expression without command specification.

1.3 Functions

Functions are special language elements which operates optional arguments and return values.

Built-in (or system) functions starts with special symbol $ with followed function name. Most system function names can be abbreviated. System functions are case insensitive.

User defined functions are created by programmer as a routine part. User functions can accept optional arguments. User functions starts with special symbols $$ with followed label name in routine contains this function. After label name foolows special symbol (^), routine name and arguments within parenthesis. Routine name can be omitted, this mean function is in the current routine.

If user function have not arguments and defined without parenthesis, this function must be called without arguments. This particular case have spacial name user defined variable accessible only to read.

User defined functions are case sensitive.

For each built-in system function MiniM compiler generates appropriate bytecode to call this function and for user defined function compiler generates code to call cpesified label, routine name, routine database and argument passing.

User defined functions can be changed any time but internal system functions defined by the MUMPS standard and MiniM specification.

In addition to MUMPS standard functions MiniM implements several extended built-in functions, which in most cases starts with symbol $ and symbol Z (system z-functions). Internal extended functions are case insensitive and in most cases can be abbreviated. Abbreviation supported is described in each function documentation.

On user function execution special system variable *$test* is stacked and protected from change on prior stack level. See reference to system variable *$test*.

1.4 Operators

Operators are special language elements, have one or two arguments (operands) and returns value evaluated dependent of operator type and over operands. Result is placed instead of operator and operands.

Operators can be used in any place when allowed expression usage.

Operators can be classified as arithmetical, logical and string operators, or classigied by programmers discretion, because official operators grouping is absent.

Operators appear as a special reserved symbol. If operator have only one argument, it is an unary operator and is applied to right operand. If operator have two arguments, it is binary operator and applies to left and right operand.

Between operator symbol and operands no any spaces are allowed.

Operator arguments, in one part, can be any expression too, including other operators with arguments.

Operators in MiniM cannot be overloaded or redefined.

Operator examples:

```
Arithmetic addition: a+b
Multiplication: a*b
Negate a number: -a
Logical not: 'a
```

1.5 Expressions

Expression is a special language part, which specify the rule how to evaluate value. Expression result in MUMPS language is a string. Expressions are constructed using operators, constants, variables and functions calls.

Expressions does not limited by complexity.

In MUMPS language unlike of other languages expressions evaluates in left-to-right order without priorities if evaluating order does not directly specified using parenthesis.

If expression is an operator result, the operator arguments can be expressions too and evaluates in left-to-right order.

If expression contains any name indirection form, this name uncovers and evaluates.

Variable subscripts also can be expressions and on variable evaluation subscripts are evaluated in left-to-right order too.

Expressions examples:

```
A+B*2
A+(B*2)
$tr(input," ")_":"_$j
^pos($j,"left")+offset
```

Expression result always is a string, but operators can use strings as a numbers or, vice versa, expression can be evaluated as arithmetic operation but result can be used as a character sequence.

1.6 Constants

Constants are the special language elements, which defines values as is.

Constants can be specified as string, number and hexadecimal constants. This is different forms how to specify byte sequence. Real values meaning are always context-dependent.

If constant is a string constant, symbols of string must be embraced to double quotes ("). It is supposed string contains symbols from open quote to closing quote and does not contain embracing quotes. If the quote symbol (") must be placed into string constant, it must be doubled. Other symbols are writes as is except nonprintable characters. String can contain a tab character too. Example:

```
s a="string"
s a="string ""quoted"""
```

Numerical constants can be specified as integers or floating point numbers. Decimal separator is a decimal point and exponent of number must be specified in a science notation with "e" or "E" exponent separator. Example:

```
s a=1234
s a=123.456e+78
```

Hexadecimal constants are specified with hexadecimal sign(#) with following hexadecimal characters case insensitive. Example:

```
s a=#123456
s a=#CAFE
s a=#BEEF
```

When routine is compiled, constants values are placed into special bytecode section of constants and used repeatedly. All places in routine with the same constant or names refers to the same place in bytecode constant section. This allow to decrease bytecode size.

Constants usage is an expression usage and constants can be used in any place where are allowed expressions. Constants cannot be assigned and passed by reference.

1.7 System variables

System (or special) variables are the special language elements which returns special or system information, have no arguments and defined by the MUMPS standard or added as MiniM extended variable.

System variables return in most cases special information about process state or any parts of process state.

System variables starts with symbol $ and have no arguments.

Extended MiniM system variables starts with symbol $ and followed symbol z.

System variables are case insensitive and in most cases can be abbreviated. For each system variable is specified possible abbreviation.

System variables are exists in any time process exists and cannot be removed. Values of system variables evaluates on call from internal process data structures.

Some system variables can be changed by direct assignment, and some variables are for read only. System variables can be used in any place where expressions are possible.

In dependent of system variable type this variable can change value indirectly, on some conditions was changed.

System variables examples:

Return current job (process) number: *$j*

Return or assign the caret position: *$x*, *$y*

Return name of last global reference: *$zr*

Create and return new GUID: *$zguid*

1.8 Structured system variables

Structured system variables are the special language elements, which are used as a special subscripted variables but does not affect any storable data. All structures and subscript values are calculated on the fly. This virtual variables exposes information about available jobs, globals, locking objects and others.

Structured system variables have special predefined names and starts with special symbols ^$.

Structured system variables names are case insensitive and can be abbreviated to one symbol after ^$.

Structured system variables are the special virtual variables and subscripts are exists if exists special appropriate objects.

This variables can be used as an arguments of system functions *$order()*, *$query()*, *$data()* and in special cases can be evaluated as value and be used as a *kill* command argument.

For each supported structured system variable in the current documentation is listed all supported operations and this operations meaning.

Values of structured system variables subscripts are dependent of variable type, for example subscripts of ^$ROUTINE are routine names, ^$LOCK are locked variables and ^$JOB - available job numbers.

If structured system variables are used as a *$order()* and *$query()* functions, this allow to get list of available system elements, function *$data()* can check element exists, *$get()* can evaluate value of element and *kill* command can remove selected element. But all this operations are dependent of ssvn type, for example, removing from ^$LOCK removes lock selected in first subscript and removing from ^$JOB terminates process selected by number in first subscript.

1.9 Local variables

Local variables are special language elements, it is automatically creted on assignment variables be a parts of current process only and are unaccessible from other processes.

Local variables does not stored on the disk.

Local variables names can starts from the special % symbol or English letter with followed optional English letter or numeral.

Local variable names are case sensitive.

Local variables can be with or without subscripts and each different name have own separate value.

Local variables have no declaration and are automatically created on first assignment.

Local variables can be created by commands *set*, *for* or *read* at the first assignment any value. Local variables are removed by the *kill* command and stacked local variables automatically are removed by the *quit* command if variable has been newed by the *new* command on this stack level.

Process can have different local variables with the same name if this variabled was newed by the *new* command on different stack levels. In this case process use only one variable with this name, nearest visible by stack.

MiniM process supports memory limiting for locals storage to prevent erroneous memory allocation. Common available local storage returns special system variable *$storage*.

Unlike of other variable types local unsubscripted variables can be passed as argument by reference. In this case formal argument as local variable is a synonym of local variable passed as actual argument. In this case local variable passed can be changed from function including subscripts.

Value of local variable in MiniM Database Server can be up to 32K.

Examples of local variables:

```
%
%X
Aks
w67
height
pos(12,"start")
```

1.10 Global variables

Global variables (or globals) are the special language elements, stored variables.

Globals exists after process created terminates and are accessible by all other processes.

Global's data and names are stored in special data files.

MiniM optimize global access and use special data caching algorithms.

All global change operations are journaled, recorded as special records in a journal file. Journal informetion is used by transaction rollback command and to restore database from backup.

Global name starts from special circumflex symbol (^), with following optional database name embraced into vertical bars and with followed global name. If global name have subscripts, subscripts must be specified into parenthesys after global name and be comma delimited. On each subscript must be specified expression to evaluate subscript value. Subscripts cannot be omitted. Database name can be omitted, this mean global in current database.

Globals in database have no any declaration and are created on the first *set* command. After removing last available data in the global this global no more exists.

Global anmes are case sensitive and database names not. Global name must be starts with special symbol % or with English letter with followed optional English letter or numerals.

Examples:

```
^GG
^|"user"|GG
^|"user"|GG($j)
^|"user"|GG("name",12)
```

For compatibility with other MUMPS implementations and with available M software MiniM supports database specification in square brackets, not only in vertical bars.

Examples:

```
^["user"]GG
^["user"]GG($j)
^["user"]GG("name",12)
```

If system function must return global name with database name, function return database name in canonical form - in upper case and embraced into vertial bars.

MiniM supports special rules for mapping globals data: 1) globals with names starts from % phisically are stored in the "%sys" database, 2) global names starts with "mtemp" phisically are stored in the "temp" database and 3) routines starts with % phisically are stored in the "%sys" database.

MiniMono difference

MiniM Embedded Edition supports only one %SYS database and all globals are stored in this one database.

1.11 Postconditional expressions

Postconditional expression is an expression evaluated befor command execution to check need this command executed or not. Expression evaluated as a number and compares with 0. Id result is 0, command does not exeuted and vice versa.

If command have several command arguments, postconditional expressin applies to all this arguments. I. e. command applies to arguments in the left-to-right order or control passed to the next command.

Postconditional checking does not affect to system variable *$test* unlike of the *if* command and postconditional expression acts only to one command and the *if* command acts to all followed commands in the line.

Postconditional expression writes after command keyword and delimited by colon, for example:

```
cmd:postcond cmdarg1,cmdarg2
```

Here *postcond* is a postconditional argument, *cmd* is a command keyword and *cmdarg* are the command arguments.

Although postconditional argument writes after command keyword, this value evaluates before command execution.

Even if the postconditional expression evaluates as a 0 and command does not executes, expression evaluation can have a side effects.

Several commands such as *if*, *else* and *for* cannot have a postconditional expressions because this commands applies not to one argument, but to all followed commands in the line.

Some commands, for example *do* and *goto*, can have postconditional expressions for an arguments too. If argument is specified without postconditional expression, if command executes, this argument executes unconditionally. Otherwise if argument specified with own postconditional expression, this expression evaluates first to check need this argument be used or not. If been specified both postconditional expression for command and for an argument, the command postconditional expression evaluates first. If command does not executed, the argument's postconditional expression does not evaluates too. For example:

```
do:pc label1:pc1,label2:pc2
```

Here if expression *pc* evaluates as 0, all arguments and argument's postconditionals are ignored. Otherwise next evaluates postconditional expression *pc1* and if it is nonzero, command applies to argument label1. Otherwise evaluates a postconditional expression *pc2* and if it is nonzero, command applies to argument label2. Otherwise control passes to the next command.

Argument postconditional expressions can have side effects too and does not affect to system variable *$test*.

1.12 Strings and numbers

MiniM Database Server conforms to current MUMPS standard to use strings and numbers. In most cases all values in MUMPS programs are strings or a byte sequence. And values can be used in special context as a numbers. In this case string is used as number representation.

If process need to use value as a number, process use only first string part that syntactically conforms to a number. String to a number casting can be arithmetic and canonic. Arithmetic casting always have a number as a casting result, and canonic casting have a number as a result only if string is canonical number representation. Canonical number representation is defined by several rules: no leading signs except optional minus, no leading and trailing zeroes, and after number no any other symbols. Arithmetic

casting accept all leading signs, leading and trailing zeroes but all symbols not in number template are ignored. And, for arithmetic casting, if any number's symbols not found, result is 0.

Arithmetic casting is made by operation or function context if operator or function use value of an argument as a number.

Canonical casting is used by name construction from subscripts values. If subscript value is a number in a canonical form, this subscript is used and sorts as a number, otherwise is used and sorts as a string.

Some examples of canonical string to number casting can be demonstrated using one (in arithmetic meaning) number:

```
TEMP>s a("0.1")="0.1"

TEMP>s a("0.10")="0.10"

TEMP>s a(".1")=".1"

TEMP>s a(".10")=".10"

TEMP>w
a(.1)=".1"
a(".10")=".10"
a("0.1")="0.1"
a("0.10")="0.10"
```

Here only variable *a(.1)* have a numeric subscript, other variables have a string subscripts.

On the other hand, if operator or a function use value as a string but value is got in arithmetic operation, process cast number to a string. In number to string casting is used canonical representation of a number. In result value have not leading plus sign, leading and trailing zeroes and have a normilized mantissa if need. Example of this casting:

```
TEMP>w 0.2*3_4
.64
TEMP>w $l(0.2*3)
2
```

1.13 Subscripts

In the MUMPS language local, global and structured system variables can be subscripted and create tree-like structures. Each node of this tree defined by name and full subscruipts set in specified order. Each this name can have separate value. Different names or different subscripted names have separate own values.

In MUMPS language supposed that if full subscripted name have no data, this name does not exists, but can have child names. Examples:

```
a(12,34)="456789"
^GL78("date")="78,89"
^GL78("date","oper",25)="127.0.0.1^80^4523"
```

Global or local subscripted names can have any values or byte sequences up to 32K length. Subscripts can have any values, strings or numbers. Common subscripts count can be up to 63 and full subscripted name can be up to 255 bytes length.

MUMPS have several built-in functions to use subscripted names in different ways: enumerate available subscripts values and names ($order() and $query()), check name and subscripts have data ($data(reference)), function to split full subscripted name to parts ($qs(name,pos)) and function to construct name from parts (set $qs(name,pos) = value).

MiniM implements common rule to sort subscript values. Much less of all is an empty string, next follow numbers in arithmetic order and next follow all other strings in alpha sorting defined in special collation file.

Current collation file is a special .N file in the /nat subdirectory. Configuration file minim.ini must contains this file name in section Server, key Locale. Collation file must be specified without extension.

Collation table applies by all processes in a common to a global names, a local names, structured system variables and to special operator "follows after". On data extraction processes use collation tables with an available variable structures created on data insertion.

On deletion global or local name by standard *kill* command MiniM removes the specified name and all subscripted names independent of count of child names are available.

1.14 Naked indicator

Naked reference (or naked indicator) is a special case of global name last called. For each system function who can call global name been defined side effect to change value of naked indicator. For example, function *$data()* changed naked indicator even if global name in argument does not have data and does not exists.

Each process have own value of naked indicator and on process start current naked indicator is an empty string.

If process change current database, value of naked indicator changes to an empty string.

Each process can evaluate own value of naked indicator using special system variable *$zreference*.

Process can change value of naked indicator by assigning system variable *$zreference* with the *set* command. New value can be an empty string or any valid global name. MiniM check syntsx of value assigned and if it is not a valid global name, generates an error <SYNTAX>.

Value of naked indicator is used by special syntax element, containing of circumflex and followed subscripts in parenthesis. For example:

```
^("date")
^(456,$h)
```

Naked indicator can be used in any place when global variable name is allowed.

On naked indicator usage real global name creates at real usage. It is much important - naked indicator translates into real global nmae not in the evaluating order, but in usage moment. For example, if naked indicator is assigned from other global variable by code

```
set ^(12)=^G(34)
```

first of all process read value from global *^G(34)*, at this moment value of naked indicator is changed, and next the *set* command use naked indicator to construct full global name to assign to. Here new name to assign is derived from previously used (*^G* in this case), but not existing before the *set* command execution.

On naked indicator using result global name is constructed by the rule: last subscript of naked indicator is replaced to first specified subscript and other subscripts are appended. For example:

```
TEMP>s $zr=$na(^a(12))

TEMP>w $na(^(34,56))
^a(34,56)
```

On naked indicator usage can be generated an error <NAKED> in the two cases: 1) value of naked indicator is an empty string and 2) value of naked indicator is global name without subscripts.

Naked indicator usage on the one hand can make code very hard to read and understand and on the other hand can simplify and make very nute to read and understand. Moreover, if naked indicator is used, need to garantee global calls order after code changed.

1.15 Indirection

Indirection is a special language element allows to use name, part of name or other syntax element evaluated at run time.

Indirection can make code unreadable and vise versa more readable and pure in dependent of usage case.

Indirection in the MUMPS language can be the following:

- name indirection
- pattern indirection
- command argument indirection
- subscript indirection
- *$text()* function indirection

If process on execution must use indirection, argument of indirection evaluates as a string and result is used as substituted into executed place instead of indirection.

Name indirection

Name indirection can be used in any place where is allowed variable name. Evaluated expression is used instead the variable name, for example:

```
USER>s abc=123,name="abc",var=@name

USER>w
abc=123
name="abc"
var=123
```

Here been used name indirection in expression *@name*. Value of expression has been evaluated as a "abc" string and this string has been used instead of variable name.

In name indirection value evaluated can contain variable name with subscripts.

Pattern indirection

When pattern indirection been used, evaluated expression is used as a pattern content. For example:

```
s pat="1N"
w 3?@pat
```

Here evaluates expression *pat* and result value is used instead pattern. Really executes pattern matching *3?1N*.

Command argument indirection

When used command argument indirection evaluated expression is used as command argument or arguments comma delimited. Command will be applied to this argument or arguments, for example:

```
USER>s value="a=123",@value

USER>w
a=123
value="a=123"
```

Here the *set* command is used with indirection form and command applies to result of expression as "a=123" and really process execute as command *s a=123*.

Subscript indirection

When subscript indirection is used, evaluated expression is used as a name with possible subscripts and to this name appends one or more specified subscripts. Result is used as a real name. For example:

```
USER>s name="a(1,2,3)",@name@(4,5,6)=123456

USER>w
a(1,2,3,4,5,6)=123456
name="a(1,2,3)"
```

Here evaluates the *name* expression as name with 3 subscripts and to this name appends again 3 subscripts and result name is used.

$text() function indirection

When *$text()* function indirection is used, argument can be specified indirectly as full argument or as any part of argument. For example:

```
s label="LABEL^RTN"
w $text(@label)
w $text(@$p(label,"^",1)^@$p(label,"^",2))
w $text(@$p(label,"^",1)^RTN)
```

Here in case 1 is used full argument indirection, in case 2 is used separate label and routine name indirection and in case 3 is used label indirection with direct routine name specification.

1.16 Routines

Routines are the special database element, which contains lines of code with commands.

Routine texts consists of line of code with commands and in first symbols must be whitespace or a label.

Routines are saves, edits, compiles, exports or imports entirely, as one object.

MiniM Database Server is an interpreter of compiler type. Process can execute only compiled bytecode. On top command line execution entire line is compiled into temporary bytecode and executes. On routine execution process executes appropriate part of this routine bytecode. For routien execution the routine source code does not required except special *$text()* function usage cases.

Routine source text is stored in the global ^ROUTINE in each database. Compiled bytecode is stored in the global ^rOBJ in the same database.

For routine names there implemented special name mapping rules for routines starts with symbol %. This routines phisically stored in the "%SYS" database and visible and can be used and executed by any process from any database.

Routine name can starts with symbol % or an English letter with followed English letter or a numeral. Routine name must be up to 31 symbols length. Routine names are case sensitive and routines with different names are different routines.

Label in the routine line can be used as subroutine name. To this label (or with offset from) there is possible to pass control by the *goto* command or call as a function with arguments passing.

Labels names are case sensitive and must have different names within one routine.

In routine can be used special language element whicj cannot be used in ordinal command line - block syntax. If line starts with dot after whitespace (after optional label without arguments), one or more this lines followed by each other are block of code. This block of code is an internal unnamed subroutine and can be executed by the *do* command. On block execution process creates next stack level and lines executes if have the same start dot count. This blocks can be included into each other and included block must have dots by one more than parent block. Block's dots can be with or without whitespaces delimited. For example:

```
label(param)
  new i
  for i=1:1:param do  quit:i>5
  . write i,!
  quit
```

Here argumentless *do* command executes subroutine as block of code as lines specified with dots. If after line of block execution next line have less then need dots, it is equal to execute hidden argumentless *quit* command and control returns from this unnamed subroutine. And in the line with argumentless *do* command after this *do* command can be other commands.

To call label as a subroutine it must be specified as label name with circumflex and routine name. If database name of routine does not specified, it is used current database. Database name can be specified embraced into vertical bars:

```
^|DbName|RoutineName,
```

where dbname - expression evaluated as a string. Database names are used case insensitive. In simple cases can be used only constant:

```
do ^|"MINIM"|Sample()
```

Label names are written before the circumflex symbol:

```
LabelName^|DbName|RoutineName
```

If the label name is omitted, this mean label in first line of routine. Between label name and circumflex can be specified a plus sign with expression with offset (mandatory positive value):

```
LabelName+Offset^|DbName|RoutineName,
```

where *offset* - any expression, evaluated as an integer and must be a positive.

If label is used with offset, there is impossible to use and pass arguments.

After routine compilation compiler creates bytecode and this bytecode is stored in the ^rOBJ global. Bytecode is stored and used as single byte sequence and cannot be greater than 32K length.

If programmer got this bytecode limit, there is possible to split routine's subroutines to several routines.

On bytecode execution MiniM Database Server uses special caching to optimize bytecode access time. One bytecode can be used by all processes in the current MiniM instance.

MiniMono difference

MiniM Embedded Edition supports only one %SYS database and all routines and compiled bytecode are stored in this one database.

1.17 Labels

Labels are special symbol sequences with label name in routine source text. Label starts from first symbol in the line of code and can starts with symbol %, English letter of numeral with followed English letters or numerals. Each label in the routine source text must differs. Label names are used case sensitive.

Label can be with or without arguments embraced in parenthesis after label name. After label name without arguments or after parenthesis must be at least one whitespace symbol. Arguments must be delimited with comma.

Label can be used by the *do* command to call label as subroutine, by call as function with return value using $$ syntax and for *goto* command to pass control to line with label. For *goto* command label must be without arguments.

Routine example with labels:

```
FUNC(a,b) q a+b*2    ; 1
%NAME q $p($zv," ")  ; 2
```

Here in first case is label FUNC with two argumenrs *a* and *b*. this label can be called as function $$FUNC(), with passing 0, 1 or 2 arguments values. In the second case is label %NAME without arguments and can be called without arguments as $$%NAME.

On the same line of code with label can be commands and before first command must be at least one whitespace symbol. On label calls as function, by *do* or *goto* commands process starts execution from first command followed after label or after label arguments defined.

How label has been called is defined on the call side, as subroutine or as a function. Subroutine called can determine what type of call been made using system variable *$quit*. If call side wait return, subroutine must return value using argumented *quit* command, otherwise simply return execution using argumentless *quit* command.

At the call side after label name can be specified offset from label and routine name containing this label. If label called with arguments or as function, offset usage is impossible. Label name, offset, routine nmae can be omitted, if other parts are enough to determina line of code. If routine name is omitted, this mean currently executed routine, if offset is omitted (in the most cases) this mean no offset, if label name is omitted, this mean first available line of routine usage.

Examples:

```
do LABEL^RTN          ; 1
goto LABEL+3^RTN      ; 2
g 12+4                ; 3
do ^RTN               ; 4
w $$FUNC^RTN($h,ver)  ; 5
```

Here in case 1 called subroutine without arguments passing in routine RTN and label LABEL. In case 2 made *goto* command to 3-rd line after label LABEL in routine RTN. In case 3 program goes to 4-th line after label 12 in current routine. In case 4 program call as subroutine first line of routine RTN. In case 5 program call as function label FUNC in routine RTN with passing two arguments.

If does not specified label and offset, process pass control to first line of routine.

On labels call label indirection can be used. Indirection can be applied to any part of full label reference. If label name is stored in variable *label* and routine name in variable *rtn*, the previous samples can be writen as:

```
do @label^@rtn          ; 1
goto @label+3^@rtn      ; 2
g @label+4              ; 3
do ^@rtn                ; 4
w $$@label^@rtn($h,ver) ; 5
```

1.18 Parameters passing

When parameters passed to a label each actual argument matches to a formal parameter inside subroutine or a function.

As as actual arguments can be used expression results, local variables passing by reference or actual argument can be skipped. If argument is a local variable by reference, it is not allowed to use subscripted name.

Count of actual argument and count of formal parameters can be different. Formal parameters matches to actual argument by position in left-to-right order. If actual arguments been passed less then accepted format parameters, other parameters have undefined values.

Formal parameters are unsubscripted local variables, which got value or be a synonym of passed by reference variable and exists on this stack level. On leaving current stack level this local variables are lost.

If actual argument is skipped, appropriate formal parameter got undefined value.

MiniM Database Server implements two extensions for MUMPS standard: 1) if formal parameter is undefined after matching, it can get special value

by default and 2) label can accept variable-lenght arguments into one format parameter.

Automatic assignment to undefined parameter

If formal parameter got undefined value after parameters matching, it can be specified what value need to be assigned to this parameter by default. After parameter name must be the equal sign ("=") with followed constant to assign.

Let it be the subroutine:

```
LABEL(v=123)
 q v
```

This subroutine can be called as:

```
$$LABEL()    ; 1
$$LABEL(2)   ; 2
$$LABEL(3,8) ; 3
```

Here in case 1 actual argument does not passed and formal parameter v got undefined value, but value by default is specified and variable v got the value of constant 123. In case 2 formal parameter accept passed value of 2. In case 3 passed more actual arguments then accepted and function call generate an error <PARAMETER>.

Variable length arguments

If label must accept unknown arguments count, label can declare last formal parameter with 3 dots:

```
LABEL(p1,params...)
```

In this case when formal parameters matched to actual arguments, parameter *params* accept count of arguments since 2 and more and subscripted values of *params* accept the values of arguments been passed. Subscripts counts from value 1 and matched to actual arguments in left-to-right order. If there been skipped one or more arguments, appropriate subscripted values will not have values (undefined value).

For example, let it be a subroutine:

```
LABEL(params...)
 zw params
 q params
```

And, we can get several variabts how to call and pass arguments:

```
USER>w $$LABEL^test(1)
params=1
params(1)=1
1
USER>w $$LABEL^test(1,2)
params=2
params(1)=1
params(2)=2
2
USER>w $$LABEL^test("we",2)
params=2
params(1)="we"
params(2)=2
2
USER>w $$LABEL^test(,2)
params=2
params(2)=2
2
```

Caution: two additional MiniM extensions are not a part of MUMPS standard and programmers must check have the target M implementation compatible parameters passing or not.

1.19 Comments

Comment is a part of line of code from comment symbol (;) to the end of line.

Before comment symbol in line of code must be at least one whitespace symbol, for example space or tab symbol.

Comment can be placed in any part of line of code where can be command keyword.

On execution MiniM threats comment as end of line.

All characters after comment symbol to end of line are ignored on execution.

If comment starts from comment symbol with following comment symbol (;;), this line of code is stored in bytecode and later is accessible by the

$text() function even if source of routine is not accessible. In this case size of bytecode is increased to store entire line.

Comment can be used in routine's line of code as such as in *xecute* argument, and in this case argument can starts from comment symbol without whitespace sequence. If comment is present in *xecute* argument, all argument after comment symbol is ignored.

1.20 Locks

Lo0cks are special internal server objects for process synchronization. Lock objects can be created and checked to existing. Lock objects have names the same as global and local variables.

Locking is made by special *lock* command. Command have as an argument local or global variable name and locking method.

For locking objects name specified does not required or used. This variable can exist and no exist and this variable access does not blocked.

MiniM use several rules for locking objects. Different processes cannot lock in one time the same names or names which are in parent-child hierarchical relation. But different processes can lock names with difference at least in one subscript value. For example, if one process lock name *a(12,34)*, second process cannot lock name *a(12,34)* and names *a* and *a(12)* and names *a(12,34,any subscripts)*. But second process can lock names which are differs by any subscript, for example last subscript: *a(12,45)*, *a(12,67)* or *a(15)* because this name have not any parent-child relation with lock made as *a(12,34)*.

All locking objects are internal server objects and are visible by all processes but owned by only one process made locking. After process terminates all locking objects been owned by this process are lost. If process terminates unexpectedly, locking objects are removed by special guardian process after this guardian rollback database changes made.

All available locking objects existing on this MiniM instance made by any process are listed in special structured system variable ^$L[OCK]. Caution: getting full locking list can take a time and locking objects can be created or removed within this time because MiniM is a multiprocess server and processes are executed concurrently.

Each locking object have own locking counter. If locking has been made first time, locking counter sets to value 1. Next time process can obtain incremental locking with the same name and locking counter is increased by one

on each incremental locking. Process also can use decremental unlocking and locking counter decrements by one. If locking counter still equal 0, locking object removes from locks entirely. Each locking object have process number who is an owner of this lock and structured system variable ^$L[OCK] return information about who is lock owner and about locking counter.

Locking can be made by special mathod, to wait list of names. In this case process wait possibility to lock all listed names and make lock for entire names list. If at least one name cannot be locked, no names from list been locked and process still wait.

Locking can be made using optional timeout specification. If timeout does not used, process can wait locking name or names infinitely. If timeout is present, process wait only whithin specified time and in timeout is expired, *lock* command return control to next command. If lock timeout has been specified, process sets special system variable *$test* to value 1 if lock created successfully or to 0 if locking does not made. Program can check this variable after *lock* command to determine been lock successful or not.

Locking is a MUMPS standard part and required to correct concurrent access to shared objects such as globals. Caution: locks used only for access synchronization, not for objects blocking.

MiniM Database Server extend standard MUMPS locking mechanism and allow to read variable ^$L[OCK] to get information about lock owner and locking counter and allow to *kill* from the ^$L[OCK] variable to remove locking object independently from real locking owner. This allow to remove the deadlock state manually.

1.21 Input-output devices

MiniM Database Server supports input-output device model as specified in MUMPS standard. Each process can open and use several devices and own at the same time. Unlike of other systems MiniM process is owner only for device been opened in this process and cannot use devices opened by other process. On process closing all devices been opened by this process automatically closed.

Devices are identified by device identification string. Each device have case-sensitive device name. Common device name consists of:

```
|DeviceType|device-type dependent string
```

All *open*, *use* and *close* use device identification strings with possible optional device options.

MiniM Database Server implemets several device type as built-in and can accept user-defined device types from external dll. For example, some of built-in device types:

CON	Process interact using standard Windows console window.
FILE	Input-output to file of file system.
NULL	Null device, all output losts and all input return empty data immediately.
PIPE	Channel to interact with child console Windows process.
STD	Standard input-output with redirection and for batch mode.
TCP	Input-output using TCP/IP sockets.
TNT	Interaction over TCP with telnet clients.

Device name string after device type specify device name to distinquish from other possible devices with the same type and this part can contain some device details. For example, for FILE device it is file name, for PIPE it is operating system command with parameters and for TCP it is server name and port number.

Some device types can be created only automatically on process start, it is CON, TNT and STD devices and automatic device creation is dependent from process run conditions.

Some device types are interactive devices and support keyboard input indirectly or over remote client (STD, CON and TNT devices). For interactive devices TNT and CON MiniM support true caret positioning and extended string input mode with built-in string editor.

One of devices of process creates automatically on process start and still active all time process exists. For example, if process has been run using the *job* command, process have by default NULL device and if process has been run directly as minim.exe, default device is a CON device. This automatically created device cannot be closed until process terminates and have special name - principal device. Device identification string of principal device return special system variable *$p[rincipal]*.

In according of MUMPS standard, all input and output are made using current device. Only one of opened by process devices can be current device.

To make device current must be used the *use* command and device still current until next *use* command change current device. Device identification string of current device return special system variable *$i[o]*.

For program compatibility with other MUMPS systems MiniM Database Server supports pseudodevice 0. Value 0 can be used as device identification string for *use* command to call principal device. And command *use 0* is synonym for *use $p*.

On closing device which is current and is not principal device, MiniM closes this device and make current default principal device. There is impossible to have absent current device.

If MiniM process goes into interactive mode to input next commands to execute, process makes principal device current automatically using hidden command *use $p*. All other opened devices still opened.

MiniM Database Server allow to different processes open and own devices with the same device identification strings, but each of this device is separate device and are owned by different processes. For example, two or mode processes can open the same file in file system to append strings or overwrite different segments and can use file locking. Or two processes can open interprocess PIPE device with the same operating system command to execute.

Full device list with device names been opened by process can be obtained using structured system variable ^$D[EVICE]. This variable show only devices been opened by current process only.

Input from device can be made only from current device and there allowed to input string or one symbol.

Output to device can be made only to current device and there allowed to output string, one symbol and use special output formatting.

The *read* and *write* commands can operate not only strings and symbols, there allowed to use special formatting output, for example line feed or form feed or clear screen. This special formatting output dependent of real device type.

The *read* command, regardless of name, can output constants and apply special formatting output to make input much useful, for example:

```
read !,"Input file name: ",fname
```

here the *read* command outputs line feed formatting, from line begin outputs string "Input file name: " and goes into real read mode to accept input as a string. Read result is placed into a *fname* local variable.

1.22 Device options

Commands *open*, *use* and *close* can accept additional options with detailed information what to do with device or change device mode or other function.

Device options are specified after device identification srtring folllowed by colon. If option is only one, option can be specified without parenthesis, otherwise parenthesis are mandatory to group several device options. In this case device options are separated by colon.

Device options can be specified in two manner - by name or by position. Device option can be with or without value. MiniM distinquish options specified by position by option position number. If option is specified by name, this underly of specifying by position. If option is specified by position, there is placed only value in approptiate position. Examples:

```
u file:"WT"           ; 1
u file:("WT")         ; 2
u file:(/MODE="WT")   ; 3
u socket:/ACCEPT      ; 4
```

Here listed different ways how to specify device option. Variant 1 - option specified by position, here option is only one and parenthesis omitted. Variant 2 - the same as 1 but option in parenthesis. Variant 3 - option is specified by name (here MODE is an option name). Variant 4 - option is specified by name and have no value.

Option names and possible values are listed in special chapter for each supported device type.

Real device option meaning is dependent of device type, and device option can change device mode as such as make some function with device. For example, device options /MODE and /TERM changes current device mode and read terminator, and device options /LOCK and /ACCEPT make file locking and accept incoming tcp connection.

MiniM Database Server implements many device operations as possible using device options to save MUMPS syntax without creating many special *$view("dev")* functions.

1.23 Device mnemonics

Mnemonics are the special language elements, special *read* and *write* arguments with possible arguments in parenthesis. Mnemonics starts with special

symbol "/" with followed mnemonic name and optional arguments in parenthesis and comma delimited. As arguments can be passed any expression result. Mnemonics seens like a procedures to handle special input-output actions. For example, to control device with caret position change to special coordinates with syntax independent of current device type.

For each device can be assigned special routine to handle all this device mnemonics. And if commands *read* or *write* have a mnemonic as an argument, process automatically transform this mnemonic call to subroutine call with optional parameters passing. Each device can have separate routine dependent of device type but call syntax can still identical for several used devices. For example, the code:

```
write /CUP(12,20)
read /CUP(12,20)
```

make caret positioning on the device in interactive mode and specify caret coordinates to position to. This code seems like device-independent, but internals of routines to handle this mnemonics can differ from each other for console and for telnet.

Mnemonic routines can be specified automatically in the server configuration file minim.ini, or can be changed or assigned at any command *open* or *use*. After mnemonic routine assignment this routine still applied to device until changed.

In the most cases mnemonic routines are organized to make special libraries with standard names to make application portability.

Each mnemonic routine must have a label for each mnemonic supported with optional arguments dependent of mnemonic usage. MiniM process automatically transform mnemonic call to subroutine call with optional parameters passing. In mnemonic routine all parameters are accepted by value only. If subroutine return control by the *quit* command, the *read* or *write* commands pass control to the next command or handle next command argument.

Chapter 2

Operators

2.1 Unary Plus (+)

Evaluates argument as a number.

Syntax

+ expr

Definition

Operator *Unary Plus* (+) evaluates argument as a number and produces result. Value of argument used as a sequence starts with number and symbols after number representetion are ignored. Number representation can have any leading zeroes, plus and minus signs, decimal point and exponent.

Leading signd are used to determine number sign. Absolute value of number does not changes. Operator *Unary Plus* may be applied to any expression.

Operator *Unary Plus* transform number into internal MiniM representation and later this value can be transformed to a string, but all automatic number-to-string conversions are made using canonical number representation and result may differs from source *Unary Plus* argument. For example, number 0 always converted to string without sign.

Examples:

```
TEMP>w +"123"
123
TEMP>w +"--123"
123
TEMP>w +"-++-+-123"
```

```
-123
TEMP>w +"-++-+-123.23"
-123.23
TEMP>w +"123.23e12"
123230000000000
TEMP>w +"123.23e1 ff"
1232.3
TEMP>w +" 123.23e1 ff"
0
```

2.2 Addition (+)

Evaluates an arithmetic sum of two operands evaluated as numbers.

Syntax

expr1 + expr2

Definition

Operator evaluate both operands as numbers and produce an arithmetic sum. If operator cannot evaluate result using available decimal digits and get numeric overflow (integer or floating point), operator generate an error <MAXNUMBER> or <MINNUMBER>.

Examples:

```
TEMP>w "--123"+"--456"
579
TEMP>w "1a"+"2b"
3
TEMP>w "a"+"b"
0
```

2.3 Unary Minus (-)

Evaluates argument as a number and reverses the sign.

Syntax

- expr

Definition

Operator *Unary Minus* (-) evaluates argument as a number, reverses the sign and produces result. Value of argument used as a sequence starts with number and symbols after number representetion are ignored. Number representation can have any leading zeroes, plus and minus signs, decimal point and exponent.

Leading signd are used to determine number sign and reverses the sign. Absolute value of number does not changes. Operator *Unary Minus* may be applied to any expression.

Operator *Unary Minus* transform number into internal MiniM representation and later this value can be transformed to a string, but all automatic number-to-string conversions are made using canonical number representation and result may differs from source *Unary Minus* argument. For example, number 0 always converted to string without sign.

Examples:

```
TEMP>w -"123"
-123
TEMP>w -"--123"
-123
TEMP>w -"-++-+-123"
123
TEMP>w -"123.23e1 ff"
-1232.3
TEMP>w -" 123.23e1 ff"
0
```

2.4 Subtraction (-)

Operator produces the difference between two operands.

Syntax

expr1 - expr2

Definition

Operator *Subtraction* evaluates both operands as numbers and produce arithmetic difference between operand values.

If operator cannot evaluate result using available decimal digits and get numeric overflow (integer or floating point), operator generate an error <MAXNUMBER> or <MINNUMBER>.

Examples:

```
TEMP>w 123-45
78
TEMP>w "12 a"-"8 b"
4
```

2.5 Multiplication (*)

Operator produces the product of left and right operands as numbers.

Syntax

expr1 * expr2

Definition

Operator evaluates both operands as numbers and produces the product of operands.

If operator cannot evaluate result using available decimal digits and get numeric overflow (integer or floating point), operator generate an error <MAXNUMBER> or <MINNUMBER>.

Examples:

```
TEMP>w 12*45
540
TEMP>w "12 r"*"45 p"
540
TEMP>w ""*""
0
```

2.6 Division (/)

Operator produce division result of left operand to right.

Syntax

expr1 / expr2

Definition

Operator evaluates both operands as numbers, divide left to right and produce result. If right operand evaluates equal 0, operator generate an error <DIVIDE>.

Division result depending of operands can be integer and floating point number. If operator cannot evaluate result using available decimal digits and get numeric overflow (integer or floating point), operator generate an error <MAXNUMBER> or <MINNUMBER>.

Examples:

```
TEMP>w 123/45
2.73333333333333
TEMP>w 123/""

<DIVIDE>
TEMP>w 123/"12ff"
10.25
```

2.7 Integer Divide (backslash)

Operator produces integer result of division two operands.

Syntax

expr1 \ expr2

Definition

Operator evaluates both operands as a numbers and produces the integer result of division the left operand by the right operand and does not return a reminder.

Examples:

```
TEMP>w 12\8
1
TEMP>w 789\123
6
```

Most widely used case this operator usage is evaluating integer part of a number:

```
TEMP>s v=123.456 w v\1
123
```

2.8 Exponentation (**)

Operator raise left operand to power selected by right operand.

Syntax

expr1 ** expr2

Definition

Operator evaluates both arguments as numbers and reise left operand to power of right operand.

Operator use available decimal digits and if result cannot be represented, operator generate errors <MAXNUMBER> or <MINNUMBER> on integer or floating point overflows.

MiniM can raise to a negative as well as to a positive power.

If value of left operand evaluates as a negative number, operator generate an error <ILLEGAL VALUE>.

Examples:

```
TEMP>w 12**-1
.0833333333333333
TEMP>w 12**-3
.000578703703703704
TEMP>w 12**-3.2
.000352062697854864
TEMP>w 4**.5
2
TEMP>w 2**2
4
TEMP>w 2**""
1
```

2.9 Modulo (#)

Operator produces the value of an arithmetic modulo operation on left and right operands.

Syntax

expr1 # expr2

Definition

Operator evaluates both operands as numbers, divide left to right and produce result as arithmetic modulo operation.

If value of right operand evaluates as zero, operator generate an error <DIVIDE>.

Examples:

```
TEMP>w 123#8
3
TEMP>w 123\8
15
TEMP>w 123#0

<DIVIDE>
TEMP>w 123#8
3
TEMP>w 123.456#8
3.456
```

2.10 Concatenation (_)

Produce concatenation of operands as strings.

Syntax

expr1 _ expr2

Definition

Operator evaluates both operands as strings and prodice new string value with concatenation of operands values. If argument is a number, operator cast number to a string using standard MiniM casting rules. Length of result string is a sum of operands lengths after casting to string.

Concatenation operator can be applied to any values but if result overflows maximum MiniM string limitation (32K), operator generate an error <MAXSTRING>.

Examples:

```
TEMP>w 123e2_"vv"
12300vv
TEMP>s a="b",b="c" w a_b
bc
```

2.11 Not (')

Inverts the truth value of the boolean operand or inverts logical operator or inverts pattern code meaning.

Syntax

' expr

left 'Operator right

'Patcode

Definition

Unary *Not* operator evaluates operand *expr* as number and compares with zero. If value is zero, operator produces value 1, otherwise produces value 0.

Unary *Not* operator is mostly wide used to cast expression value to canonical boolean value using two operators: if expression evaluates as 0, result of two *Not* operators is 0, otherwise is 1.

Examples:

```
TEMP>w '""
1
TEMP>w '"asd"
1
TEMP>w ''"asd"
0
TEMP>w ''123
1
```

Second form of operator *Not* is applicable to logical comparision operators to invert their logic. Table of operators which can be used with *Not* operator and result operator:

'=	Not equal.
'¿	Not greater (less or equal).
'¡	Not less (greater or equal).
'[	Not contains.
']	Not follows.
'&	Not AND.
'&&	Not AND (used lasy AND).
'!	Not OR.
'——	Not OR (used lasy OR).
'?	Not match pattern code.

Examples:

```
TEMP>w 123'>12
0
TEMP>w 123'=12
1
TEMP>w 123'<12
1
```

If *Not* operator is applied to other operators, compiler generate an error <SYNTAX>.

The *Not* operator with logical comparision operator X can be transformed to unary *Not* operator by formula:

```
a 'X b == '( a X b)
```

If *Not* operator used in third form and is applied to pattern code, operator produce pattern code negating. If pattern code match to "this symbols", *Not* operator with pattern code produce matching to "not this symbols". For example:

```
TEMP>w "MiniM"?1"M"1.'N
1
```

Here pattern matching specify the rule: one symbol "M" and followed one or more non-digit symbols.

2.12 Equals (=)

Operator check is operands values are equal.

Syntax

expr1 = expr2

Definition

Operator evaluates both operands as strings anb compares byte-to-byte. If both strings are equal, operator prodice value 1, otherwise produce value 0. Operator compare strings case sensitive.

If one or both operand values are numeric, MiniM can cast this values to strings with bytes different from originally wrote by programmer, MiniM use canonical number-to-string casting rules.

Examples:

```
TEMP>w 0123=123
1
TEMP>w "0123"="123"
0
TEMP>w +"0123"=+"123"
1
TEMP>w "123.0"=123.0
0
TEMP>w +"123.0"=123.0
1
```

2.13 Greater (¿)

Operator compares operands as numbers.

Syntax

expr1 ¿ expr2

Definition

Operator evaluates both operands as numbers and compares as numbers. Result is 1 if left operand is greater than left, otherwise result is 0.

Examples:

```
TEMP>w "a">"b"
0
TEMP>w "0123">"123"
0
TEMP>w "01230">"123"
1
TEMP>w "2 apples">"1 apple"
1
```

2.14 Greater or Equal (¿=)

Operator compares operands as numbers.

Syntax

expr1 ¿= expr2

Definition

Operator evaluates both operands as numbers and compares as numbers. Result is 1 if left operand is greater than left or equal left, otherwise result is 0.

Operator *greater or equal* is not part of MUMPS standard and intended to simplify code writing and meaning. It is equal to operator *not less*.

Examples:

```
TEMP>w "a">="b"
1
TEMP>w "0123">="123"
1
TEMP>w "01230">="123"
1
```

2.15 Less (¡)

Operator compares operands as numbers.

Syntax

expr1 < expr2

Definition

Operator evaluates both operands as numbers and compares as numbers. Result is 1 if left operand is less than left, otherwise result is 0.

Examples:

```
TEMP>w "a"<"b"
0
TEMP>w "0123"<"123"
0
TEMP>w "01230">"123"
0
TEMP>w "01230"<"12300"
1
```

2.16 Less or Equal (¡=)

Operator compares operands as numbers.

Syntax

expr1 <= expr2

Definition

Operator evaluates both operands as numbers and compares as numbers. Result is 1 if left operand is less than left or equal left, otherwise result is 0.

Operator *less or equal* is not part of MUMPS standard and intended to simplify code writing and meaning. It is equal to operator *not greater*.

Examples:

```
TEMP>w "a"<="b"
1
TEMP>w "012"<="013"
1
TEMP>w 123<=45
0
```

2.17 Contains ([)

Operator produce truth value if left operand contains right operand as strings.

Syntax

expr1 [expr2

Definition

Operator evaluates both operands as strings and search right operand in left operand case sensitive. If it is found, operator produce 1 otherwise 0.

Special case of the *Contains* operator is right operand equals an empty string. In this case operator always produce 1, it is a part of MUMPS standard.

Examples:

```
TEMP>w 123[12
1
TEMP>w "MiniM"["N"
0
TEMP>w 123456[""
1
```

2.18 Follows (])

Operator tests operands follows in ASCII collating sequence.

Syntax

expr1] expr2

Definition

Operator evaluates both operands as strings and tests value of left operand follows after right operand in ASCII collating sequence. To compare symbols operator compare ASCII codes of symbols.

Examples:

```
TEMP>w 123]""
1
TEMP>w 123]123
0
TEMP>w ""]123
0
TEMP>w "minim"]"mini"
1
TEMP>w 1]0
1
```

2.19 Sorts After (]])

Compares two operand using subscript sorting.

Syntax

expr1]] expr2

Definition

Operator *Sorts After* tests whether left operand sorts after right using current subscript collation sorting. Operand evaluates both operands and check is it numbers or strings and use subscript collation and rules to produce result. Result is 1 if left operand sorts after right operand.

Subscript collation rules are: first of all sorts empty strings. After empty strings sorts numbers as numbers in arithmetic order. After numbers sorts other strings using current server collation defined in minim.ini file, section Server, key Locale.

The same rule is used to sort variable's subscripts for global, local and structured system variables.

2.20 AND (&)

Evaluates a logical AND operation of operands.

Syntax

expr1 & expr2

Definition

Operator evaluates both operands as truth-values (zero or not) and produce binary AND operation using the following table:

expr1	expr2	AND
0	0	0
0	1	0
1	0	0
1	1	1

Examples:

```
TEMP>w 123&456
1
TEMP>w "a"&"b"
0
```

2.21 Lazy AND (&&)

Evaluates a logical AND operation of operands.

Syntax

expr1 && expr2

Definition

Operator *Lazy AND* evaluates left expression as truth value. If result is 0, operand does not evaluate right operand and produces result 0. Otherwise operator evaluates right operand as truth value and if it is 0, produce result 0, otherwise produce result 1.

Operator *Lazy AND* is not a part of MUMPS standard and if goal is write portable programs it is not recommended to use. Operator have side effect dependent of left operand evaluation result.

Examples:

```
TEMP>k

TEMP>w 0&&$i(a),! w
0

TEMP>w 1&&$i(a),! w
1
a=1
```

Here in first case operator have no side effect of *a* variable incrementing, and in second case have.

If operator *Not* is applied to operator *Lazy AND*, it saves lazy behavior.

2.22 OR (!)

Evaluates a logical OR operation of operands.

Syntax

expr1 ! expr2

Definition

Operator evaluates both operands as truth-values (zero or not) and produce binary OR operation using the following table:

expr1	expr2	OR
0	0	0
0	1	1
1	0	1
1	1	1

Examples:

```
TEMP>w 1!"a"
1
```

```
TEMP>w " "!"a"
0
```

2.23 Lazy OR (||)

Evaluates a logical OR operation of operands.

Syntax

expr1 || expr2

Definition

Operator *Lazy OR* evaluates first left operand as truth value and if it is 1, does not evaluate right operand and produce result 1. Otherwise operator evaluates right operand as truth value and if it is 1, produce result 1, otherwise result is 0.

Operator *Lazy OR* is not a part of MUMPS standard and if goal is write portable programs it is not recommended to use. Operator have side effect dependent of left operand evaluation result.

Examples:

```
TEMP>w 1||$i(a),! w
1

TEMP>w 0||$i(a),! w
1
a=1
```

If operator *Not* is applied to operator *Lazy OR*, it saves lazy behavior.

2.24 Pattern Matching (?)

Operator produces result is string matched to pattern or not.

Syntax

expr1 ? expr2

Definition

expr1	Expression evaluates as string to match pattern.
expr2	Pattern.

Right part *expr2* can be evaluated on execution if it is used pattern indirection:

```
expr1 ? @expr2
```

In this case MiniM use *expr2* evaluated as string as pattern.

MUMPS pattern consists of sequence of atomic patterns followed by each other. One atomic pattern consists of repeat counter with following pattern code or symbol sequence or alternation:

```
pattern = repcount patcode
pattern = repcount string
pattern = repcount alternation
```

Repeat counter consists of digits with or without dot. Repeat count is applied once to the following pattern code and mean matched string contains this specified symbols with this repeat count. Repeat counter can be specified using the following syntax or repeatition:

count	Only this repeat count.
.	Any count including 0.
min .	At least *min* times, maximum is not limited.
. max	Maximum *max* times, minimum can be 0.
min . max	From *min* to *max* times inclusively.

Pattern code is a special symbol to specify symbols class:

A	Letter.
C	Nonprintable symbols. Symbols with ASCII code from 0 to 31 and 127.
E	Any symbol.
L	Letter in lower case.
N	Digit.
P	Punctuation symbols and space.

U	Letter in upper case.

Pattern codes can be use case insensitive.

To determine symbol class as letter MiniM process use server current national collation table specified in minim.ini file, section Server, key Locale. This file is a special symbols tables edited by MiniM Collation Editor. File store information how to collate symbols and how to make upper and lower case. MiniM consider that symbol is a letter if conversion to upper and lower case have different results.

Pattern code can follow one by one, and this group specify united symbols class. For example, pattern code UN is "symbol in upper case or digit". For example:

```
TEMP>w "MiniM"?1UN.E
1
```

MiniM can apply the *Not* operator to pattern code to invert symbols class, result of class inverting is a new class "not in this class". For example:

```
TEMP>w "MiniM"?1'N.E
1
TEMP>w "MiniM"?1'NP.E
1
```

Here first pattern specify matching rule: first symbol is not a digit, with eny following and second rule: first symbol is not a digit and not punctuation with any following.

Symbol sequence is pecified by ordinal MUMPS string following by *pat-count* without spaces and ebreced to double quotes, for example:

```
TEMP>w "MiniM"?1"M".E
1
```

Here specified matching rule: first one string "M" with any following symbols.

Other samples:

```
TEMP>w "MiniM"?1"Mini".E
1
TEMP>w "MiniM"?1"Minim".E
0
```

This patterns specify matching rules for strings starts with "Mini" and with "Minim".

Pattern alternation is specified by parenthesis with alternative patterns delimited by comma:

```
alternation = ( pattern [ , pattern ...])
```

Alternation pattern specify matching to one of alternative.

Examples:

```
TEMP>w "MiniM"?1(1"Min",1"Max").E
1
TEMP>w "MiniM"?1(1"mos",1"ber").E
0
```

Alternation pattern can be specified within other alternation.

With pattern alternation can be specified special symbol pseudo classes, for example matching to octal digits:

```
TEMP>w "MiniM"?.(1"0",1"1",1"2",1"3",1"4",1"5",1"6",1"7")
0
TEMP>w "0177"?.(1"0",1"1",1"2",1"3",1"4",1"5",1"6",1"7")
1
```

And, using pattern indirection this new symbol class can be used shortly:

```
TEMP>s patoct=".(1""0"",1""1"",1""2"",1""3"",
         1""4"",1""5"",1""6"",1""7"")"
TEMP>w "MiniM"?@patoct
0
TEMP>w "0177"?@patoct
1
```

2.25 Hexadecimal (#)

Operator return integer from his hexadecimal representation.

Syntax

 # symbols

Definition

Operator *Hexadecimal* is a part of MiniM numbers syntax, is used on compilation stage and cannot be applied to evaluated expression. After symbol # must follows hexadecimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F). Result of interpreting this hexadecimal digits sequence is a operator result.

Less significant digit is specified by rightmost hexadecimal symbol, and leftmost symbol specify more significant digit. If count of hexadecimal symbols is odd, MiniM left pad symbols with leading zero.

When you use the hexadecimal representation must take into account special symbol # is also *write* command special format. To use hexadecimal numbers in *write* argument it is need to place hexadecimal notation into parenthesis.

Examples:

```
TEMP>s a=#41

TEMP>w
a=65

TEMP>w $a(#41)
54
TEMP>w $c(#41)
A
TEMP>w $c(#41,#42,#43)
ABC
TEMP>s a=#123,b=#ff

TEMP>w
a=291
b=255

TEMP>w (#cafe)
51966
TEMP>w (#CAFE)
51966
TEMP>w (#AAA)
2730
TEMP>w (#0AAA)
```

```
2730
TEMP>w *#41
A
```

Chapter 3

Commands

3.1 CLOSE

Command close specified device or all devices.

Syntax

C[LOSE][:pc]

C[LOSE][:pc] closearg[,closearg2,...]

C[LOSE][:pc] dev[:param]

C[LOSE][:pc] dev[:(param,...)]

C[LOSE][:pc] @indclosearg

Description

pc	Postconditional expression.
dev	Device identification name.
params	Command parameters.
closearg	Command arguments as dev[:params].
indclosearg	Argument indirection.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

If *close* command applies to principal device, it does nothing and no errors

are generated. The *close* command can close only devices which have been opened only by current process and cannot close devices opened by other processes.

While executes, the *close* command closes devices, free used by this device resources and remove device from current device list. If this device has been current, command make current device the principal device.

The *close* command without arguments closes all opened devices and make current device the proncipal device.

If specified in argument device has not been opened, command does nothing.

The *close* command arguments are specified by name or by position. If command's parameter is only once, it can be used without parentheses, otherwise parentheses are mandatory. For example:

```
s dev="|FILE|c:\temp\1234.tmp"
close dev:/DELETE                              ; 1
close dev:(/DELETE)                            ; 2
close dev:(/TRUNCATE:/RENAME="L:\1234.log")    ; 3
```

In this first case used only one parameter, it may be specified without parentheses. In the second case one parameter specified in parentheses, in last case used two parameters, and here parentheses are mandatory.

The command's parameters names are case-insensitive. For example::

```
close "|FILE|c:\temp\1234.tmp":(/DELETE)
close "|FILE|c:\temp\1234.tmp":(/Delete)
close "|FILE|c:\temp\1234.tmp":(/delete)
```

Here the /DELETE parameter is specified in different cases, but logically all cases are identical

If was specified command argument indirection, the *indclosearg* evaluates as string and the *close* command applies to the result. For example:

```
s fname="c:\tmp\123.tmp"
s opt="(/DELETE)"
...
s indclosearg="""|FILE|"_fname_""":"_opt
close @indclosearg
```

In the command indirection case the value of the *indclosearg* cannot be an empty string, in this case is generated the error <SYNTAX>.

3.2 DO

Command call the subroutine.

Syntax

D[O][:pc] entryref[:pc1][,entryref[:pc2]]

D[O][:pc] entryref([param[,param2...]])[:pc1][,entryref([param[,param2...]])[:pc2]]

D[O][:pc] @doarg

D[O][:pc]

Description

pc	Postconditional expression.
entryref	Reference to call subroutine.
doarg	Argument indirection, expression with command arguments.
param	Actual parameter (skipped, call by value or call local name by reference).

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

The *entryref* points can be in the forms::

```
LabelName[+Offset][^RoutineName]
^RoutineName
```

Here are

LabelName	Label name in the routine.
Offset	Line offset.
RoutineName	The routine name.

The routine name can be used with or without database specification. If the database name not specified, here used routinein the current database. The database specification need to be before the routine name, for example:

```
LabelName^|dbname|RoutineName
```

The *offset* evaluates as an expression, casts to the integer and need to be positive value. Here the plus symbol is part of the *entryref* syntax.

All parts of the *entryref* can be specified indirectly:

```
@LabelNameExpr[+Offset][^@RoutineNameExpr]
^@RoutineNameExpr
```

The postconditional expressions with *entryref* evaluates, if specified, as integers. If postconditional expression with *entryref* does not specified, the *do* command call subroutine, If postconditional expression is specified, it evaluates and compare with 0. If expression is 0, the *do* command does not call the subroutine, and if non zero, do.

The command argumens calls in left-to-right order.

In the subroutine context execution command *do* creates new stack level and sustem variable $quit has value 0.

The number of parameters passed can be other than subroutine accepts. Here applies common parameter passing rule, see this documentation topic about parameter passing.

If used command indirection then *doarg* evaluates as string and value used as *entryref*. The *doarg* value must be syntax-correct, it can be one or more *entryref*, delimited by comma.

Argumentless *do* command executes, if postconditional expression evaluates as non-zero, executes the block of code that immediately follows by current line. The block of code is command line's sequence with dots in first. The nested block of code need to have one more dots. The dots count is a block of code nesting level. The argumentless *do* command executes nested block of code if lines have one more dots than in the current line..

If lines of nested block of code ends, there exicutes implicit *quit* command, and nested block execution terminates. After execution argumentless *do* command the control continues with next command. For example:

```
label(param)
  new i
  for i=1:1:param do
  . write "i = ",i,!
  . f j=1:1:3 d
  . . w "j = ",j,!
  quit
```

If inside nested block of code present and executes argumentless *quit* command, it terminates block execution and control continues after the argumentless *do* command.

If inside nested block of code present and executes the *quit* command with argument, it generates the <COMMAND> error because the *do* context execution does not allow value return.

If the argumentless *do* command executes without current routine context it generates the <SYNTAX> error.

If executes argumentless *do* command there value of system variable $test is stacked, but for argumented form *do* command not. See comments for system variable $test. It is provided for comatibility with MUMPS program written prior the MUMPS have block of code.

Indirection of the DO command allows soft label recognition, and are used only first labels. All other symbols after labels are ignored.

3.3 ELSE

Executes follows commands if system variable $test is 0.

Syntax

E[LSE]

Description

The *else* command have not postconditional expression.

The *else* command compares system variable $test with 0. If value is 0, control continue execution follows commands.

The *else* command does not change value of system variable $test.

The value of $test changes on execution of *if* command, *lock*, *read*, *open* and *job* commands if timeout is specified.

3.4 FOR

Executes next followed commands as many times as specified in *for* arguments.

Syntax

F[OR]

F[OR] lvn=forparams[,lvn2=forparams2,...]

Description

lvn	Local variable name, the cycle variable.
forparams	*For* arguments, conditions to execute cycle.

The *for* command does not have postconditional expression and indirection form.

Argumentless *for* command continue execution following in line command until argumentless *quit* command, *goto* command is reached, or if the process is interrupted. If reached *quit* command with arguments, it is generates <COMMAND> error. The *quit* and *goto* commands interrupts execution of last executed *for* command if in line present several *for* commands.

The *forparam* parameters need to be specified with comma delimiting:

```
forparams = forparam[,forparam...],
```

where each of the *forparam* can have one of the following form:

expr	One value.
start:step	Cycle start and cycle step.
start:step:end	Cycle start, cycle step and cycle end.

Each of the *expr*, *start*, *step* and *end* expressions evaluates only onve before following commands executes and next variable's changes does not affect to the *for* conditions.

While executes, the *for* command determines current cycle form and compares the cycle variable with specified conditions. The cycle executes with current value of cycle variable.

If it is specified several *forparam* delimited by comma, ones applies in left-to-right order and can be specified different *forparam* forms.

One Value

If specified one value for cycle variable, the *for* command assign cycle variable to specified expression value and executes cycle once. For example:

```
TEMP>f i=1 w
i=1
```

With specifying several expressions there is possible to make cycle with special values, for example:

```
TEMP>f i="red","green","blue" w i,!
red
green
blue
```

Cycle start and cycle step

If specified cycle start and step form for *for* command, the command evaluates *start* and *step* values and use it within cycle iteration. The values of *start* and *step* does not evaluates later.

First cycle iteration executes with cycle variable assigned to the *start* value. Next iteration the *for* command increment cycle variable as number to value of *step*. Cycle execution continues until *quit* or *goto* commands reached. For example:

```
TEMP>f i=1:1 w i,! q:i>=12
1
2
3
4
5
6
7
8
9
10
11
12

TEMP>
```

The cycle variable value as such as other local variable can be changed inside cycle. If after next iteration the cycle variable have undefined value, the *for* command generates the <UNDEFINED> error. For example:

```
TEMP>f i=1:1 w i,! k:i=5 i q:i>=12
1
2
```

```
3
4
5

<UNDEFINED>
TEMP>

TEMP>f i=1:1 w i,! s i=i+$random(5) q:i>=12
1
2
5
6
8
11
12
```

The cycle direction, incrementing or decrementing the cycle variable, determined by the *step* variable.

Cycle start, step and end

The *for* command behavior in this form depends of the *step* value sign, is it negative or non-negative.

If the *step* is non-negative, the *start*, *step* and *end* values evaluates as numbers, and next this does not change and used as it evaluates. The cycle variable assigned to the *start* value. If the variable value is greater than *end* value (compared as numbers), the cycle is done. Otherwise executes commands followed by the *for* command.

After next cycle iteration the *for* command compare the cycle variable with the *end* value. If it is greater than *end*, cycle terminates. Otherwise the *for* command increment cycle variable by the *step* value. Result compares with the *end* again and if it is greater, cycle terminates. Otherwise following commands executes. And this iterations repeates again. For example:

```
TEMP>f i=1:2:8 w i,!
1
3
5
7

TEMP>
```

If the *step* is negative, the *start*, *step* and *end* values evaluates as numbers, and next this does not change and used as it evaluates. The cycle variable assigned to the *start* value. If the variable value is less than *end* value (compared as numbers), the cycle is done. Otherwise executes commands followed by the *for* command.

After next cycle iteration the *for* command compare the cycle variable with the *end* value. If it is less than *end*, cycle terminates. Otherwise the *for* command increment cycle variable by the *step* value (negative). Result compares with the *end* again and if it is less, cycle terminates. Otherwise following commands executes. And this iterations repeates again. For example:

```
TEMP>f i=3:-2:-6 w i,!
3
1
-1
-3
-5

TEMP>
```

So, the greater - less comparisions with the *end* value depends of the *step* sign and programmer can change the cycle variable to change cycle execution. For example:

```
TEMP>f i=1:1:5 w i,! i i=3 s i=10
1
2
3

TEMP>
```

3.5 GOTO

The *goto* command continue execution from the specified line.

Syntax

G[OTO][:pc] labelref1[:pc1][,labelref2[:pc2],...]

G[OTO][:pc] @gotoarg

Description

pc	Postconditional expression.
labelref	The label name, offset, routine name.
@gotoarg	Argument indirection with label(s).

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applued to all arguments. Command applies to argument in left-to-right sequence as specified.

Postconditional expressions with labels evaluates to check need to go this label, If postconditional expression is absent, *goto* command go to specified label. Otherwise postconditional expression evaluates and compares with 0, if it is zero, *goto* command ignores label and continues execution or check next specified *labelref* delimited by comma.

The *goto* command go to label only once, and check in left-to-right order.

The full *labelref* can contain label name, offset and routine with database specification. If routine name is absent, used label in the current routine. If label name is absent, line counts from first line of routine. If offset is absent, execution continues from the line with label. If label name and offset does not specified, execution continues from first line of routine.

Offset need to be (if present) as expression and cannot have negative values.

The *goto* command does not pass any parameters to *labelref*.

To execute *goto* from the block of code command skip lines until reached line with the same (or less) dots count as the line at which *goto* is.

If specified non existing *labelref*, the *goto* generate error about label does not exist.

If specified indirection command form the *gotoarg* need to be syntax-correct *labelref* list. If indirection is specified, the value of *doarg* is evaliated first and then *goto* command use this string.

Indirection of the GOTO command allows soft label recognition, and are used only first labels. All other symbols after labels are ignored.

3.6 HALT

The *halt* command terminates process execution.

Syntax

H[ALT][:pc]

Description

pc	Postconditional expression.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

The *halt* command, depending of server settings, executes transaction rollback or commit, or nothing, closes all opened by process devices, unlock all locked local and global variables, frees all resources and loaded dynamic libraries. For example:

```
h       ; 1
h:a=1   ; 2
h:a     ; 3
```

Here in 1 case executes unconditional process exit, in 2 case exit executes in depending variable a is equal 1 and in 3 case executes exit if variable a is defined and is nonzero.

MiniMono difference

In the MiniM Embedded Edition the halt command terminates current host process call but does not terminates the entire process. After MiniMono call was terminated by the halt command, host process can call MiniMono again. Host process must terminate semself.

3.7 HANG

The *hang* command suspend process execution to specified number of seconds.

Syntax

H[ANG][:pc] time[,time2,...]

H[ANG][:pc] @hangarg

Description

pc	Postconditional expression.
time	Expression to specify timeout to suspend.
hangarg	Argument indirection.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

The *hang* command evaluates the *time* expression as number. The result is used as seconds number, and command suspend process execution. The *time* is used as number with fractional part and precision up to milliseconds. For example:

```
hang 2   ; 1
set t=2  ; 2
hang t
```

Here in first case process hangs to 2 seconds and timeout is specified by constant and in second case timeput is specified by expression.

In command argument indirection case the value of *hangarg* evaluates as string and used as *hang* arguments. For example:

```
h @("3,3")
```

Here one-by-one executes two *hang* commands by 3 seconds.

3.8 IF

The *if* command continue execution of the followed commands if command's condition is non-zero.

Syntax

I[F]

I[F] expr[,expr2,...]

I[F] @ifarg

Description

expr	Truth value expression.
ifarg	Argument indirection.

The *if* command have not postcondition expressions.

The *if* command in argumentless form tests the $test value. The $test value compares with 0 and if $test value is not 0, *if* command continue execution of the followed commands in current line. Otherwise command execution at this line terminates, and acts as line end.

The *if* command with argument evaluates argument and compares with 0. If it is nonzero, *if* command continue execution of the followed commands in current line. Otherwise command execution at this line terminates, and acts as line end.

The *if* command with indirection evaluates argument as string and it used as command argument.

If the *if* command have several arguments delimited with comma, the *if* command evaluates argument sequientially in left-to-right order and compare each argument with 0.

Examples:

Check the value of $test variable and make action dependent of value:

```
TEMP>w $t
0
TEMP>i  w 1

TEMP>
```

Evaluate expression and make action dependent of value:

```
TEMP>i 1,2 w 123
123
TEMP>s a=0

TEMP>i a w 456

TEMP>
```

Evaluate arument indirection form and apply command to result:

```
TEMP>s ifexpr="1,2"

TEMP>i @ifexpr w 123
123
TEMP>s ifexpr="1,0"

TEMP>i @ifexpr w 123

TEMP>
```

As *ifexpr* expression can be used any MUMPS variables, functiona and operators.

If the *if* command continue execution, the $test variable sets to 1 value, otherwise sets to 0. Later the $test value can be used with argumentless *if* command and with *else* command.

3.9 JOB

The *job* command run new MUMPS process.

Syntax

J[OB][:pc] jobargument,...

```
jobargument =
  label^|database|routine(params):(process-params):timeout


process-params =
  process-param[:process-params...]
```

J[OB][:pc] @jobarg

Description

pc	Postconditional expression.
jobargument	New job label start specification.
label	Label in routine to start new job from.
database	Database of routine to start new job from.
routine	Routine name to start job from.
params	Actual passed parameters.
process-params	New job parameters, environment.
timeout	Timeout to wait new job starts, seconds.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

In the *job* parameters passing here does not allowed passing parameter by reference to local variable. Other passing methods are allowed like as ordinal subroutine call. If *job* command reached passing local variables by reference it generates <JOB> error.

There is not allowed to specify label offset and parameters simultaneously.

The *job* command runs new MiniM process, which starts execution from the specified label in specified routine. Process get the passed parameters *param*. New process have the value of $zparent system variable with job number of parent process. After execution the parent process have in $zchild system variable number of child process. This value is available until the next *job* command executes and does not expired if child process terminates.

New child process is running in separate Windows address space, have own local variables, devices and locks. All MiniM processes executes concurrently. If the computer have several CPU, processes can executes concurrently in the same time.

New child process have principal device |NULL| by default.

If timeout is specified, the *job* command sets the system variable $test to 1 if child process runs successfully and otherwise to 0. If timeout does not specified, the value of $test dows not affected.

New child process can get additional variables as process parameters (environment). The process parameters need to separated by comma.

First process parameter (if present) show the database name to run child process. Child process switches to specified database on start. If process parameter is only once, the parentheses are not mandatory.

The second process parameter is the device name with type |TCP| to pass to child process. For this device need to be executed the /ACCEPT operation. Child process gets the tcp/ip socket to use concurrently and child process have principal device |TCP| by default. Device has options to read and write and work in bynary mode. Child process can change device options using the *use* command with principal device.

The third process parameter need to be (if present) the string with local variables names delimited by comma. This variables with values as is are created inside child process on start. It is full copy including indexed values with the same names.

Any of process parameters can be omitted, but parameter position need to be specified by colons. Before last omitted process parameters colons are not mandatory. All process parameters can be passed as constants as cuch as expression. All expression are evaluated in left-to-right order.

New child process start execution with "DO" context and after quit from subroutine child process terminates.

One *job* command can have several arguments, for each of one command runs new child process as specified in left-to-right order.

The *jobarg* value evaluates as a string and used as *job* arguments, and can contain one label to run one child process or several to run several child processes.

MiniMono difference

MiniM Embedded Edition does not implement the JOB command.

3.10 KILL

The *kill* command removes local or global variables including subscripts (if exists).

Syntax

K[ILL][:pc]

K[ILL][:pc] varname

K[ILL][:pc] ssvn

K[ILL][:pc] killname[,killname2,...]

K[ILL][:pc] (lockvarname1[,lockvarname2,...])

K[ILL][:pc] @killarg

Description

pc	Postconditional expression.
varname	Local or global variable name.
ssvn	Structured system variable if one allow *kill* command.

killname	Local or global or structured system variable if one allow *kill* command.
lockvarname	Local variable name.
killarg	Argument indirection.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

Argumentless *kill* command removes all local variable which are visible at the current stack level. It does not mean removing all other variables with the same name on other stack level. If variable has been created at level 1 and newed on the level 2, the *kill* command removes only variable at the 2 level and does not affect to level 1.

Removing local or global variable

If specified local or global variable name, this variable removes including all available subscriupts. If variable is a global variable, it makes an appropriate journal records, and *trollback* command can rollback removing later. If removes local variables, it does not journaled. For example:

```
kill ^abc
kill a,b,c
```

If specified variable has been locked bu the *lock* command, the *kill* command ignores lock. This relate to local and global variables.

Removing structured system variable allowed only in special cases of this variables. Each structured system variable has own special meaning what does the *kill* command. For example, kill from ^$LOCK affect removing lock, and kill from ^$JOB affect killing MiniM process. For details what allowed with *kill* command and structured system variables see reference to selected structured system variable.

Exclusive *kill* command removes all local variables except specified and listed in parentheses delimited by comma. In list allowed only unsubscripted local variable names. Global and structured system variables does not allowed in exclusive *kill* command. For example:

```
TEMP>s a=1,b=2,c=3,d=4

TEMP>w
a=1
b=2
c=3
d=4

TEMP>k (b,d)

TEMP>w
b=2
d=4

TEMP>
```

Argument indirection for *kill* command evaluates expression *killarg* as a string and *kill* command applied to result. The *killarg* content need to be a valid *kill* arguments. For example:

```
TEMP>s a=1,b=2,c=3,d=4

TEMP>s killarg="b,d"

TEMP>k @killarg

TEMP>w
a=1
c=3
killarg="b,d"

TEMP>
```

If variable does not exist, the *kill* command does nothing.

3.11 KSUBSCRIPTS

The *ksubscripts* command removes local or global variable subscripts if ones exists.

Syntax

KS[UBSCRIPTS][:pc]

KS[UBSCRIPTS][:pc] varname

KS[UBSCRIPTS][:pc] killname[,killname2,...]

KS[UBSCRIPTS][:pc] (lockvarname1[,lockvarname2,...])

KS[UBSCRIPTS][:pc] @killarg

Description

pc	Postconditional expression.
varname	Local or global variable name.
killname	Local or global variable name.
lockvarname	Local variable name.
killarg	Argument indirection.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

Argumentless *ksubscripts* removes subscripts in all available on the current stack level local variables. It does not mean removing all other variables with the same name on other stack level. If variable has been created at level 1 and newed on the level 2, the *ksubscripts* command removes only variable at the 2 level and does not affect to level 1.

The *ksubscripts* command removes only subscripts, it not affected to the specified name. For global variable the *ksubscripts* command make journal records and *trollback* command can rollback removing later. If removes local variables, it does not journaled. For example:

```
ksubscripts ^abc
ksubscripts a,b,c
```

If specified variable has been locked bu the *lock* command, the *ksubscripts* command ignores lock. This relate to local and global variables.

The *ksubscripts* command cannot be allplied to structured system variables, this case generate the <COMMAND> error. it does not relate to structured system variables, it is specific of MiniM implementation.

Exclusive *ksubscripts* command removes all local variables except specified and listed in parentheses delimited by comma. In list allowed only unsubscripted local variable names. Global and structured system variables does not allowed in exclusive *ksubscripts* command. For example:

Argument indirection for *ksubscripts* command evaluates expression *killarg* as a string and *ksubscripts* command applied to result. The *killarg* content need to be a valid *ksubscripts* arguments.

If variable or subscripts does not exist, the *ksubscripts* command does nothing.

3.12 KVALUE

Removes the specified variable without removing subscripts.

Syntax

KV[ALUE][:pc]

KV[ALUE][:pc] varname

KV[ALUE][:pc] killname[,killname2,...]

KV[ALUE][:pc] (lockvarname1[,lockvarname2,...])

KV[ALUE][:pc] @killarg

Description

pc	Postconditional expression.
varname	Local or global variable name.
killname	Local or global variable name.
lockvarname	Local variable name.
killarg	Argument indirection.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

Argimentless form of the *kvalue* command removes all local variables, visible at the current stack level without removing subscripts. Argumentless *kvalue* command does not remove variables with the same name at another

stack level. If variable has been created at level 1 and newed on the level 2, the *kvalue* command removes only variable at the 2 level and does not affect to level 1.

Removing local or global variable

If specified local or global variable, it removes except subscripts. If database is joutnaled, removing values make a journal records about each removed variable. After this the *trollback* command can roll back variable's values. Local variables are not journaled. For example:

```
kvalue ^abc
kvalue a,b,c
```

If specified variable has been locked bu the *lock* command, the *kvalue* command ignores lock. This relate to local and global variables.

The *kvalue* command cannot be allplied to structured system variables, this case generate the <COMMAND> error. it does not relate to structured system variables, it is specific of MiniM implementation.

Exclusive *kvalue* command removes all variables except specified in the list and except subscripts. In the list can be specified only local variable names.

Argument indirection for *kvalue* command evaluates expression *killarg* as a string and *kvalue* command applied to result. The *killarg* content need to be a valid *kvalue* arguments.

If variable does not exist, the *kvalue* command does nothing.

3.13 LOCK

Create or remove lock of local or global variable.

Syntax

L[OCK][:pc]

L[OCK][:pc] glvn[:to]

L[OCK][:pc] glvn[:to][,glvn2[:to2],...]

L[OCK][:pc] (glvn1[,glvn2...])[:to]

L[OCK][:pc] +glvn[:to]

L[OCK][:pc] +(glvn1[,glvn2...])[:to]

L[OCK][:pc] -glvn

L[OCK][:pc] @lockarg

Description

pc	Postconditional expression.
glvn	Local or global variable name.
lockarg	Argument indirection.
to	Timeout to wait lock.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

The *lock* command create or remove lock of local or global variable. Lock is an internal MiniM Database Server object, it is visible by all existing MiniM process of one instance. Lock have name of the local or global variable.

The local or global variable does not need to exist. The database of lock does not exist. Database name does not checked on existence and can be used database names with any names including impossible. If database name for global variable does not specified, lock mean the current database name of current process.

All existing lock objects are available in the structured system variable ^$LOCK. Each lock can be removed not only by *lock* command, can be used the *kill* command with ^$LOCK variable and lock name as first subscript. The *lock* command can unlock only owned locked variables, and *kill* command with ^$LOCK variable can remove lock independently of the lock owning.

All locks created by process autonatically are removed if process halts. If process is killed not the regular stuff, MiniM Database Server automatically run the guardian process to remove all lock objects made by this process. In this case all locks are removed only after transaction rollback to allow make correct concurrent data access.

Locking rule is to not allow two or more process lock the same name, lock the name with more subscripts or with less subscripts together. The name resolving id made for fully specified names after substituting database name (for globals) and all subscripts. Two or more MiniM processes can lock the same names which differs only by last subscripts.

If locking cannot be created, the *lock* command goes MiniM process to waiting lock until locking rule not reached ot locking timeout does not expired. The *lock* timeout evaluates as a number of seconds with milliseconds presicion. The end of locking counts from begin *lock* command. If timeout does not specified, it mean indefinite timeout to wait lock collision. Here programmers should make decision what to do to barring deadlocks.

If the *lock* timeout is specified, MiniM process sets the system variable $test, otherwise the $test does not changed. If timeout is specified and lock is successed before timeout expired, the $test variable sets to 1, if lock does not created before timeout expired, the $test variables sets to 0.

After unlocking independenly of the way to ublock, the locked name (and name with or without subscripts) as available to lock by other process within the same MiniM instance.

All MiniM instances have independent locking area and does not affect to each other.

If lock is created inside a transaction context ($tlevel'=0), lock object stiil active until the end of transaction even if process make unlock. On the transaction end process check lock objects made within transaction and unlocked and really remove lock objects. If lock object is removed by kill ^$LOCK variable, it remove lock object immediately and independently of transaction contest of the owner.

Argumentless form of *lock* command

```
lock
```

Argumentless form of the *lock* command removes locking of all owned lock objects, independently of count and presence of owned locks.

Name locking

```
lock glvn
```

As the lock name can be specified local or global variable names, and with subscripts. Global names can be specified with or without database name. If database name does not specified, it is used name of current database. Here MiniM applies mapping rules - if global variable name starts from the percent symbol (%), used the "%sys" database, if name starts with mtemp, used the "temp" database.

Before execution the *lock* command removes all owned locks, even if locking name already is locked. After this make lock for specified name. For example, while executes the code

```
lock a,b,c
```

the *lock* command removes all available and owned locks, make lock for variable *a*, remove lock for *a*, make lock for *b*, remove lock for *b* and make lock for *c*. After execution process owns only lock of variable *c*, not the *a* or the *b* variables.

With locking name the lock count increments by one.

List locking

```
lock (glvn1,glvn2,glvn3)
```

Here before locking the *lock* command unlock all owned locks and wait the moment to lock all listed names simultaneously. If it is possible, and allowed by locking rules, process creates lock for all listed names and the *lock* return execution to next command. For list locking can be locked all listed names or none of ones.

Each name in list obtain locking count to 1.

Incremental locking

```
lock +glvn
```

Incremental locking does not remove any available lock objects and increments lock count by one for specified name. If this name does not already locked, command make lock and set locking count to 1, otherwise locking count is incremented by one.

Back operation to decrement locking counter id decremental locking or incremental inlocking. If name was not locked, it doe not anything, if name was locked, command decrements locking counter by one and if locking counter couns as 0, command make unlocking name:

```
lock -glvn
```

For unlocking operation timeout does not supported.

Process can decrement locking count even for non-locked names, it is not error and process does nothing.

If process unlocks by one the name locked inside a transaction, the locking counter is decremented, but real lock object removing made on transaction end only ($tlevel = 0).

List incremental locking

```
lock +(a,b,c)
```

List incremental locking is a combination of list locking and incremental locking. Process wait possibility to lock listed names and lock and increment locking counter by one for all specified names or for none of ones.

With list incremental locking process does not remove any of owned locking objects.

Back operation for list incremental locking is list decremental locking:

```
lock -(a,b,c)
```

Here process decrements by one locking counter (if lock present) for all listed names. If lock object got zero locking counter, process unlock this name.

Argument indirection

With argument indirection the *lockarg* is evaluated as string and content used as the *lock* command argument. The *lockarg* need to be one of the *lock* possible argument, except empty string.

3.14 MERGE

Command copy data and subscripts from one local or global varibale into another.

Syntax

M[ERGE][:pc] glvn1=glvn2[,glvn3=glvn4,...]

M[ERGE][:pc] @mergearg

Description

pc	Postconditional expression.
glvn	Local or global variable name.
mergearg	Argument indirection.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional

expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

The *merge* command copy data with subscripts from local or global variable *glvn1* into local or global variable *glvn2*. The *merge* command cannot use structured system and system variables or expressions.

As source and target variable names can be specified subscripted names.

The *merge* command copy data with subscripts from source into target name from specified source name and continues for every present subscripted name. The variable's tree appends into target variable with replacing target subscripted values if subscripts are equals. For example:

```
TEMP>s a(1)=1,a(1,2)=12,a(1,3)=13,a(1,2,4)=124

TEMP>s b(2)=2,b(2,1)=21,b(2,2)=22,b(2,4,5)=245

TEMP>w
a(1)=1
a(1,2)=12
a(1,2,4)=124
a(1,3)=13
b(2)=2
b(2,1)=21
b(2,2)=22
b(2,4,5)=245

TEMP>merge b(2)=a(1)

TEMP>w
a(1)=1
a(1,2)=12
a(1,2,4)=124
a(1,3)=13
b(2)=1       ; replaced
b(2,1)=21    ; become unchanged
b(2,2)=12    ; replaced
b(2,2,4)=124 ; inserted
b(2,3)=13    ; inserted
b(2,4,5)=245 ; become unchanged
```

The *merge* command does not remove any subscripts from target name.

It is very important to understand to use the *merge* command to make data's copy.

For an argument indirection the value of *mergearg* evaluated as string and need to be the *merge* valid argument:

```
glvn1=glvn2
```

The *merge* command does not allow to merge data from variable's tree into herself, in this case is generated the <COMMAND> error. This operation is forbidden by the MUMPS standard. This case is checked even if specified merging trees into local variables passed by reference. The *merge* command can copy data into the same variable only if target differs from source by last subscript. For example:

```
USER>s a(1,2)="1.2"

USER>m a(2)=a(1)

USER>w
a(1,2)="1.2"
a(2,2)="1.2"
```

The *merge* command does not requires that source or target variables exists. If source variable does not exists, the *merge* command does nothing. If target variable does not exist and source exist, the merge command creates specified target variable.

The source and target names can be specified as names using any available forms including any name indirection forms.

3.15 NEW

Creates new variable on the current stack level.

Syntax

N[EW][:pc]

N[EW][:pc] lvn

N[EW][:pc] lvn[,lvn2,...]

N[EW][:pc] (lvn1[,lvn2...])

N[EW][:pc] lvn=expr[,lvn2=expr2,...]

N[EW][:pc] @newarg

Description

pc	Postconditional expression.
lvn	Unsubscripted local variable name.
expr	Any valid expression.
newarg	Argument indirection.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

The *new* command create on the current stack level place for the specified local name. On each stacl level can be variables with the same names and have divverent values and subscripts. If execution leaves stack level, all variables was created by *new* command are leaved and removed, and the process storage get back to use.

The *new* command creates the place for variable without assigning the value by default and evaluating this variable have resalt the <UNDEFINED> error. All next assignmens to this variable don't touch variables with same name on other stack levels. If the *new* command applies to the already newed variable, this variable's data and subscripts are removed and variable already have undefined value and no subscripts. Here is exception - the *new* command with initialization, to make *new* operation with assignment.

The *new* command can use only unsibscripted local variables. The *new* command cannot operate by structured system variables or system variables or globals except the $etrap and $estack system variables.

Argumentless *new*

```
new[:pc]
```

Argumentless *new* command removes all local variables, created on the current stack level and make current stack level as place for all newly created local variable even ones will created without *new* command.

New with name

```
new[:pc] var1[,var2...]
```

If the name of local variable is specified, the *new* command allpies only to specified name. If this variable has the data, all data and subscripts are lost and this name places on the current stack level and have undefined value. Next operations with this name do with this variable instance. If specified several local variables names, thi *new* applies to each of ones in left-to-right order. For example:

```
new a,b,c
```

is equivalence of the code

```
new a new b new c
```

The difference is only in postconditional axpression usage. The commands

```
new:expr a,b,c
```

and

```
new:expr a new:expr b new:expr c
```

are differs only by *expr* evaluating - in the first case *expr* evaluates only once and in the second - for each *new* command. The expression evaluation can have side effects and program execution can be dependent of expression.

The *new* command can be applied to list of any argument form except argumentless form.

Exclusive *new*

```
new[:pc] (a,b,c)
```

Exclusive *new* command make this stack level the place for all other local variables except specified in list. Command removes all local variables are visible at the current stack level. All this variables saves stack place but lost all data and get undefined values. All other variables autometically creates on the current stack level.

Exclusive *new* command cannot be splitted into several *new* commands. This codes:

```
new (a,b,c)
new (a) new (b) new (c)
```

are not equivalence, because after second line execution will be lost all variables, and after first line will be lost all except listed (a, b, c).

New with initialization

```
new var=expr
```

This *new* command's form is not a part of the MUMPS standard and is MiniM Language extension. This command is combination of the ordinal *new* and *set* commands. Unlike of the *set* command, this *new* command can assign value of expression only to unsubscripted local variable.

The *new* with initialization can be used together with $etrap system variable.

Argument indirection

```
new[:pc] @newarg
```

The *new* command with argument indirection evaluates expression *newarg* as string and content used as *new* argument. The *newarg* content must be valid *new* argument except argumentless form.

New with special system variables

```
new[:pc] svn
```

The *new* command can be applied to special system variables:

$etrap	Code which is invoked if error occurs.
$estack	Counts stack level from last *new $estack*.

Only this two system variables can be used as a *new* command argument.

With *new $etrap* on the current stack level created place and value for $etrap. Last call to $etrap to read and write are with this place. If the variable $etrap is not newed on the current stack level, calls to $etrap are made to previous stack level recursively.

The $etrap variable can be used with *new* with initialization.

If the *new* command applied to *$estack* variable, *$estack* sets to 0 and incremented by 1 on each new stack level is created. Later the value of $estack can be cleared again. On return to previous stack level MiniM process restores previous *$estack* value.

3.16 OPEN

The *open* command creates new input-output device and opens one with specified parameters.

Syntax

O[PEN][:pc] dev[:(params)][:to][:mnemonicspace]

O[PEN][:pc] dev...[,dev2...,...]

O[PEN][:pc] @openarg

Description

pc	Postconditional expression.
dev	Device name.
params	Device parameters, listed with comma.
to	Timeout.
mnemonicspace	Expression with routine name to handle mnemonics.
openarg	Argument indirection.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

The *open* command creates and opens specified in the *dev* expression device and insert into current opened by process device list. This device does not makes the current device. If device with this name has already been opened, the *open* command does not anything and does not change any device parameters.

To check the specified device has been opened can be checked structured system varible ^$DEVICE.

MiniM support opening by different process devices with the same name. For example, several processes can open the same file to read. One process

can open several devices with the same type, for example, several TCP/IP devices. MiniM process cannot open device with the same name as already opened. For example, to open the same file twice need to specify file name in different cases at least in one symbol.

If timeout is specified, process changes the value of *$test* variable. If process opens successfully, *$test* sets to 1, otherwise to 0. If timeout omitted, the value of *$test* does not changed. Timeout measures in seconds with precision in milliseconds. Timeout counts from command execution start moment.

MiniM process prepare actions dependent of device type. If device type is not supported, process generate <DEVICE> error.

For example, code opens file with parameters by default and with mnemonic handler routine:

```
open "|FILE|c:\temp\dat.txt":::"MNSPACE"
```

If need to specify two or more device parameters, parentheses are mandatory and if only one, not.

If used argument indirection, the value of *openarg* need to be a valid *open* arguments. Empty string does not allowed. The *openarg* can contain several *open* arguments delimited by comma.

Detailed *open* parameter specifications are dependent of device type and listed in special chapter.

3.17 QUIT

Command quits current execution context.

Syntax

Q[UIT][:pc]

Q[UIT][:pc] expr

Q[UIT][:pc] @quitarg

Description

pc	Postconditional expression.
expr	Any expression.
quitarg	Argument indirection.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

The *quit* command terminates current execution context. New execution context is created on top level while executes commands (in telnet, console, execute line-by-line file lines), with execution *do*, *for*, *xecute* command and with evaluating used-defined function ($$-context).

The *quit* command terminates execution of the last *for* command in the same command line.

The *quit* command terminates the *xecute* command execution and other commands of *xecute* does not reached.

The *quit* command cannot support several arguments, accepted only first specified.

The *quit* command with indirection evaluates the *quitarg* value as expression and this value is returned.

If current execution context is argumentless *do* command and block of code, the *quit* command quits to next command after this *do* command.

If current execution context is user-defined function, the *quit* command terminates function execution and value of *quit* argument used as return result of this function.

Argumentless *quit* command does not return anything, *quit* with argument return this argument as result. *Quit* with argument cannot be used in *do*, *xecute* and *for* context. This cases generates a <COMMAND> error.

Programmer can check need to return value in the *quit* command by checking system variable *$quit*. If *$quit* has value 0, value return does not need.

On top-level execution context, for example in teknet or in console, *quit* with argument is allowed, but value is lost.

Examples:

```
TEMP>f i=1:1:5 w i,! q:i=3
1
2
3
```

```
TEMP>x "w 12,! q  w 34"
12

TEMP>
```

3.18 READ

Command reads data from device and stores it in variable.

Syntax

R[EAD][:pc] glvn[#len][:to]

R[EAD][:pc] *glvn[:to]

R[EAD][:pc] /mnemonic[(params)]

R[EAD][:pc] ?intexpr

R[EAD][:pc] constexpr

R[EAD][:pc] format

R[EAD][:pc] readarg[,readarg2,...]

R[EAD][:pc] @indreadarg

Description

pc	Postconditional expression.
glvn	Local or global variable name to place data to.
len	Count of bytes to read.
to	Timeout to wait input data..
constexpr	String constant.
mnemonic	Mnemonic name to read using it.
format	Formatting symbol.
readarg	One of read argument form.
indreadarg	Argument indirection.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

The *read* command accepts bytes from current input-output device. Command can read several bytes and one symbol, can output data and format output.

The *read* command can have optional parameters to specify wait timeout and length limit.

The *read* command use current device terminator to detect end of read. If terminator reached in input byte sequence, command end read and terminator does not placed into read result.

After end of read command places into special system variable *$key* last input.

For example, by default most of used devices have text mode and read terminates by entering ENTER symbol. Here into *$key* placed the code 13, if reached ESCAPE key, placed code 27.

The *read* command can use or not specified timeout, it is dependent of the device type.

If read timeout is specified, it is evaluated as number and counts as a seconds with milliseconds. The wait timeout starts from *read* command begin.

If timeout is specified, the *read* command sets the *$test* value indicating read has complete before timeout expired. If read terminates by timeout ends, the *$test* sets to 0, otherwise to 1.

Read length and timeout are optional, but if specified it need to be in order length and then timeout. If length read not specified and terminator or input end are not reached, the *read* command reads up to maximum string length, 32 Kb. If timeput is not specified, *read* command can wait input unlimited time, or ends if terminator or input end are reached.

Read string

```
read glvn
```

If local or global variable is specified as *read* argument, command accepr input and place data into this variable as string. The read is executes using current device-type specific actions, specified timeout and read limit, read mode, and terminators.

As *glvn* can be used local or global variable and with or without subscripts.

Read one symbol code

```
read *glvn
```

If specified special symbol "*" before local or global variable, the *read* command reads only one byte and places code into this variable as number. If timeout expired, used code -1. For example:

```
TEMP>r *ch
w
TEMP>w
ch=119
```

Input symbol code counts from 0 to 255 inclusively.

If read timeout is specified, command sets the *$test* value, if read successful, sets to 1, otherwise to 0.

Tabbed output

Tabbed oitput is specified by special symbol "?" with following expression, which evaluates as integer. It is defined position in line to shift to. For example, this is supported by MiniM console and telnet devices. Other devices support tabbed output dependent of device type.

The tabbing expression evaluates only as integer and specifies the position in line and device need to position current output pointer. Depending of device type it can be spaces or device-special positioning. MiniM counts current position in line (as it is possible) using special system variables *$X* and *$Y*.

If evaluated tabstop expression is less then or equal zero, tabbing does nothing. Otherwize device outputs appropriate spaces to shift to specified position. The position can be over the line size, and output continues from the next line. Next output continues from resulting position. For example:

```
r #!!!,?15,"Enter name: ",name
```

This code clears the screen, outputs three empty line, shift caret to position 15 in line, outputs the string "Enter name: " and wait string to place to local variable *name*.

Mnemonic usage

If mnemonis is specified, the name should be used as is, without expression or indirection. Tha values passed to mnemonic, are passed by value only, not by reference. The mnemonic name need to be preceded by slash "/". For example:

```
read /CUP(10,20),answer
```

Here command calls mnemonic with name CUP in current mnemonic handling routine and pass two parameters.

If device have not assigned mnemonic routine, mnamonic calling generates the <MNEMSPACE> error, MDC code 26. If program call mnemonic not supported by mnemonic routine, it generates <NOLINE> error.

Output constant string

If the *read* command parameter is constant string, this string outputs to current device. For example:

```
TEMP>r "Enter your name: ",name
Enter your name: John
TEMP>w
name="John"

TEMP>
```

Here the string "Enter your name: " outputs first and *read* command accepts next input into the *name* variable.

Formatting

To format output there can be used following symbols:

#	Clear the screen, flush buffers, form feed, or other device-specific action to make the similar output clearing.
!	Output the line feed symbol.

For text mode devices MiniM outputs bytes $c(13),$c(10), and for binary mode devices outputs $c(10) only.

Formatting symbols can be delimited by comma, but it is not mandatory, For example:

```
read #!!!
```

here first clears the screen, reposition caret to first line and outputs three empty lines.

Argument indirection

The *read* command can accept argument indirection. The *indreadarg* should be expression, it evaluates as string and content is used as *read* command arguments. Empty string does not allowed. Argument indirection can be use as such as other arguments with comma delimited.

3.19 SET

Assigns the value to variables.

Syntax

S[ET][:pc] setleft=expr[,setleft2=expr2]

S[ET][:pc] (setleft[,setleft2...])=expr

S[ET][:pc] @setarg

Description

pc	Postconditional expression.
setleft	Assinment target.
expr	Expression to evaluate and assign the value.
setarg	Argument indirection.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

The *set* command evaluates expression *expr* and assigns the result to specified *setleft* target. If *setleft* are listed more than one, the *expr* expression evaluates only once and targets assigns in left-to-right order as listed. For example:

```
TEMP>s (a,b,c)=123

TEMP>w
a=123
b=123
c=123

TEMP>s (a,b,c)=$i(d)
```

```
TEMP>w
a=1
b=1
c=1
d=1
```

The *set* command can use as *setleft* target the following:

lvn	Local variable name, with or without subscripts.
gvn	Global variable name, with or without subscripts.
$KEY	System variable $KEY.
$X	System variable $X.
$Y	System variable $Y.
$DEVICE	System variable $DEVICE.
$ECODE	System variable $ECODE.
$ETRAP	System variable $ETRAP.
$ZTRAP	System variable $ZTRAP.
$extract()	Function $extract() with local or global variable as first argument.
$list()	Function $list() with local or global variable as first argument.
$piece()	Function $piece() with local or global variable as first argument.
$qsubscript()	Function $qsubscript() with local or global variable as first argument.
$bit()	Function $bit() with local or global variable as first argument.

Local variable assignment

If *setleft* is a local variable name, the *set* command writes data to specified local name. If name does not exist, it is created, otherwise overwritten.

Here can be specified local names with or without subscripts.

The *set* command use local variable place, defined by the *new* command and on the specified by *new* command stack level.

Local variable name can be specified using name indirection.

Global variable assignment

If *setleft* is a global variable name, the *set* command writes data to database. If this variable does not exist, it is created, otherwise overwritten.

The global variable name can be specified with database name. If the database name is omitted, the *set* command use the current database.

The global name can be specified with or without subscripts, and assignment does not affect other subscripts.

The global name can be specified using name indirection or using naled indicator syntax.

If journaling for current process and for database of this global name is enabled, process make a journal record about this assignment and previous value. This *set* operation can be rolled back later bu *trollback* command.

Side effect of global name assignment is changing of naked indicator. Current value of the naked indicator is available as global name in system variable *$zreference*.

System variable *$key* assignment

When *$key* is assigned, it affect only for current input-output device only. When current device is changed, *$key* value changes value too.

System variables *$X* and *$Y* assignment

When system variables *$x* and *$y* assigned, the value of evaluated *expr* expression counts as integers and writes to *$x* and *$y*. Side effects are dependent of device type. The *$x* and *$y* changes are affected only for current device. Generally, *$x* and *$y* converts to caret position change. For example, if it is MiniM console window, caret position on window is changed, and if it is telnet device, MiniM sends to client special escape sequences to reposition caret. Actual caret changes are dependent from telnet client.

System variable *$ecode* assignment

When system variable *$ecode* assigned, this value stores in special variable *$ecode*. Previous value of *$ecode* be lost.

After this into sustem variable *$zerror* assigned string "<ECODETRAP>" and process generate error <ECODETRAP>.

If *$ecode* assigned by empty string, no any errors are generated, the value of *$ecode* got empty string and value of *$zerror* does not changes.

System variable *$etrap* assignment

When the system variable *$etrap* assigned, it is done on the stack level, where this variable has created by the *new* command, or on the top stack level. Previous value of this variable on this stack level is lost.

On execution the content of system variable *$etrap* is used as code line to execute when error occurs to handle error. On siisgnment the syntax of *$etrap* content does not checked.

System variable *$ztrap* assigmnent

System variable *$ztrap* assignment is a synonym to assign to *$etrap* and make *new* command to *$estack* and *$etrap*.

The error handler assigned to the *$ztrap* variable is not independent error handler, but it is alternate method to assign to *$estack* and *$etrap* variables. Error handler assigned to *$ztrap*, executes by *goto* command. When *$ztrap* assigned, process analize content and make appropriate assignment to *$estack* and *$etrap* variables in dependent of *$ztrap* content.

Content of *$ztrap* can be label or label with leading "*" symbol.

If *$ztrap* assigned to label only, it is equals to execute the code below:

```
new $estack
new $etrap
set $etrap="g:'$es "_$ztrap
```

On error it make stack unrolling to stack level where *$ztrap* assigned and make *goto* command to specified label.

When the first symbol is "*", it is equal to execute the code below:

```
set $etrap="g "_$e($ztrap,2,$l($ztrap))
```

On error this code executes *goto* command on the stack level where the error is occured.

For example:

```
USER>s $zt="err^errhandler"

USER>w $et
g:'$es err^errhandler
USER>s $zt="*err^errhandler"

USER>w $et
g err^errhandler
```

If *$ztrap* value is not on ow allowed, process generates the error <SYNTAX>.

Function *$list()* assignment

When the function *$list()* is assigned, process assign data into variable which is the first argument of *$list()*, and this content makes a list structure.

If this variable has been undefined, it creates, if not, overwrites and used as list structure.

When used 1-argument form of *$list()* function, it makes changes of the first list element or one is created.

When used 2-argument form of *$list()* function, it makes changes of the list element with position specified in second argument. If second argument is 0, function change first element. if second argument is -1, function change last element. The specified list element after assignment gets defined value with *expr* content.

If specified element show to position over the end of list in variable, all middle elements are created and gets the undefined value.

When used 3-argument form of *$list()* function, it make replace sublist of source list structure from position specified in the 2 argument by position specified in the 3 argument. If 2-nd argument is -1, it replace sublist from last element, and 3-rd argument is ignored.

If second argument is less then -1, assignment generates a <RANGE> error.

When assignment determines the variable content is not valid list structire, it generates the <LIST> error.

When assignment determines that the full list content after assignment need to be more than 23 Kb? assignment does not made anf generated the <MAXSTRING> error.

Examples:

```
TEMP>s $li(list)=2

TEMP>f i=1:1:$ll(list) w $lg(list,i),!
2

TEMP>k  s $li(list,4)="a"

TEMP>f i=1:1:$ll(list) w $lg(list,i,"def"),!
def
```

```
def
def
a

TEMP>s $li(list,2,3)=$lb("w","r")

TEMP>f i=1:1:$ll(list) w $lg(list,i,"def"),!
def
w
r
a
```

So, to remove list elements from n to m, need to be used 3-argument form:

```
TEMP>s list=$lb(1,2,3,4) s $li(list,2,3)=""

TEMP>f i=1:1:$ll(list) w $lg(list,i,"def"),!
1
4
```

And, to make the list element undefined, in need to be replaced by list with one undefined element:

```
TEMP>s list=$lb(1,2,3,4)

TEMP>s $li(list,3,3)=$lb()

TEMP>f i=1:1:$ll(list) w $lg(list,i,"def"),!
1
2
def
4
```

If variable is a global variable, assignment make changes of naked indicator value.

As first argument can be specified local or global variable name, with or without subscripts.

Function *$piece()* assignment

When the function *$piece* is assigned, it makes changes in variable specified in the first argument. It make a string with specified in 2-nd argument delimiter, 3 argument show what piece number of string need to replace, and 4 argument show position of last piece to replace. If variable contein less then need pieces, it padded by empty string with specified delimiter.

If variable has undefined value, it creates and counts as empty string, otherwise overwrites.

As first argument can be specified local or global variable name, with or without subscripts.

Examples:

```
TEMP>s str="a,b,c" s $p(str,",")="d" w str
d,b,c

TEMP>s str="a,b,c" s $p(str,",",2)="d" w str
a,d,c

TEMP>s str="a,b,c" s $p(str,",",1,2)="d" w str
d,c

TEMP>s str="a,b,c" s $p(str,",",8)="d" w str
a,b,c,,,,,d
```

Function *$piece* can use as delimiter any string, empty, one or more symbols. For example:

```
TEMP>s str="a,,b,,c" s $p(str,",,",2)="d" w str
a,,d,,c

TEMP>s str="a,,b,,c" s $p(str,"",8)="d" w str
a,,b,,cd
```

If variable is a global variable, assignment make changes of naked indicator value.

Function *$extract()* assignment

When the *$extract()* assigned, it make changes in variable specified in first argument. 2-nd and 3-rd arguments are optional and specifies symbol position to replace from and to. The assigned value can be other length of substring to replace, and it make axpanding or collapsing string.

If 2-nd argument is not specified, it counts as 1, and replaces first symbol in the string. Otherwise replace makes from specified position.

If 3-rd argument is not specified, it counts as equal to 2-nd argument and makes changes of one symbol. Otherwise it show last position to replace.

If 2-nd argument is negative, or 3-rd argument is less than 2-nd, function assignment does not make changes and naked indicator does not change.

If before assignment this variable has undefined value, this variable creates.

If need to replace substring after last available symbol, source string is padded by space symbols as need.

Examples:

```
TEMP>s str="abc",$e(str)=1 w
str="1bc"

TEMP>s str="abc",$e(str,-2)=1 w
str="abc"

TEMP>s str="abc",$e(str,1,2)=1 w
str="1c"

TEMP>s str="abc",$e(str,5)=1 w
str="abc 1"

TEMP>s str="abc",$e(str,5,8)=1 w
str="abc 1"
```

If variable is a global variable, assignment make changes of naked indicator value.

As first argument can be specified local or global variable name, with or without subscripts.

Function *$qsubscript()* assignment

When the *$qsubscript()* function is assigned, it make changes in variable specified in first argument. Here variable content counts as name of variable. Assignment make changes in specified in the 2-nd argument part of this name. If 2-nd argument is -1, it mean the variable name is a global name and replaces the database name. If 2-nd argument is 0, it replace the variable

name, otherwise it is subsctipt number. If original name has number of subscripts less than need, assignment adds need subscripts as empty strings.

Examples:

```
TEMP>s name=$na(abc(1,2,3))

TEMP>s $qs(name,4)=4

TEMP>w
name="abc(1,2,3,4)"

TEMP>s $qs(name,0)="def"

TEMP>w
name="def(1,2,3,4)"

TEMP>s $qs(name,-1)="ggg"

TEMP>w
name="^|"ggg"|def(1,2,3,4)"

TEMP>s $qs(name,8)="8"

TEMP>w
name="^|"ggg"|def(1,2,3,4,"","","",8)"
```

If arguments are incorrect variable name or unsupported subscript number, this assignment generates a <FUNCTION> error.

If variable is a global variable, assignment make changes of naked indicator value.

As first argument can be specified local or global variable name, with or without subscripts.

Function *$bit()* assignment

When the function *$bit()* is assigned, it must have first argument a local or global variable name. Assignment threats value as a bitstring structure, and changes in this bit string one bit. Bitstring is a structure with special format and can use compression as need. If name is a global name and for current process and database journaling is enabled, it makes a journal record about bit change and about previous bit value. Record contain only logical

value and count undefined or empty string, or coll outside bitstring that bit has logical value 0.

The 2-nd argument is a bit position and should be between 1 and 262104 inclusively. Assignable expression casts to integer and compares with 0. If it is 0, function sets logical bit 0, otherwize logical bit 1.

If before assignment variable has undefined value, it creates. If variable contains data not in bitstring format, it generates an <INVALID BIT STRING> error. If specified unsupported bit position, assignment generates error <RANGE>.

Function *$bit()* assignment made as atomic operation, no other process can change value between current process reads and writes bit value.

If changes are made inside transaction context, the *trollback* command can roll back logical bit state changed in transaction and does not roll back changes made by other processes.

Examples:

```
USER>s $bit(abc,3)=1

USER>s $bit(abc,5)=1

USER>f i=1:1:10 w i," : ",$bit(abc,i),!
1 : 0
2 : 0
3 : 1
4 : 0
5 : 1
6 : 0
7 : 0
8 : 0
9 : 0
10 : 0

USER>w $bitcount(abc)
5
USER>w $bitcount(abc,1)
2
USER>w $bitcount(abc,0)
3
```

Here makes write into undefined variable one logical bit, to position 3 and after this to position 5. After this makes read logical bit values from 1 to 10. For bits who does not sets to 1, it counts as 0 bits. And common bit count is 5, 1-bits are 2 and zero bits are 3 by total.

If variable is a global variable, assignment make changes of naked indicator value.

As first argument can be specified local or global variable name, with or without subscripts.

The *set* argument indirection

When used argument indirection, the value of *setarg* is evaluated as a string and used as the *set* command argument. This string need have one or more valid *set* command arguments in the form:

```
setleft = expr
```

The *setleft* expression can be only on of allowed.

The *setarg* can containt argument indirection too, recursively. For example:

```
TEMP>s setarg="a=$i(b)"

TEMP>s @setarg

TEMP>w
a=1
b=1
setarg="a=$i(b)"
```

3.20 TCOMMIT

Makes transaction level completed.

Syntax

TC[OMMIT][:pc]

Description

pc	Postconditional expression.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

The *tcommit* command make a journal record to confirm transaction level and decrements transaction level.

If transaction level is 0, command generates the <COMMAND> error.

If transaction level after command is 0, tha changes made cannot be rolled back. Process make journal record about all transaction levels are complete and internal transaction sequence number is changed.

To commit all transaction levels it need to call *tcommit* command with *$tlevel* times repeatedly.

Command does not support arguments or argument indirection.

3.21 TROLLBACK

Roll back all globals changes made in transaction.

Syntax

TRO[LLBACK][:pc]

Description

pc	Postconditional expression.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

The *trollback* command rolls back all global changes made within a transaction and process changes current transaction sequence number.

All globals changes rolled back to state before transaction starts. States counts as phisical or logical states dependent of made operations.

The *trollback* command does not rolls back changes made by *$increment* function.

If before execution *trollback* command current transaction context is 0 (no transaction state), this command does nothing.

MiniM does not supports the *trollback* parameters. If command parameter is specified, process generates <UNIMPLEMENTED> error on execution.

Argument indirection does not supported too.

3.22 TSTART

Command creates new transaction or new transaction level.

Syntax

TS[TART][:pc]

Description

pc	Postconditional expression.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

The *tstart* command starts new transaction level. Current transaction level available in the *$tlevel* system variable. On process start transaction level is 0 and increments with each *tstart* command. Non-transaction context (*$tlevel=0*) is supported for compatibility with prior writte MUMPS programs, does not used transactions.

Each transaction call increments value of *$tlevel* by one. Maximum transaction level supported by MiniM is 255. If program try to exeed thi limit, process generates the <TLEVEL> error. For example:

```
TEMP>f i=1:1:300 ts

<TLEVEL>
TEMP>w $tl
```

```
255
TEMP>tro

TEMP>w $tl
0
TEMP>w i
256
```

The *tstart* command makes a special journal record about current process number, transaction level and current transaction sequence number. Each process and transaction have unique transaction sequence number to identify different journal records. It is used by *trollback* command and automatic database restore process on server start.

MiniM does not support *tstart* arguments and argument indirection. On execution command with argument process generates <UNIMPLEMENTED> error. For example:

```
TEMP>ts

TEMP>w $tl
1
TEMP>tro

TEMP>w $tl
0
TEMP>
```

3.23 USE

Applies parameters to device and make device current.

Syntax

U[SE][:pc] dev[:param][,dev2[:param2],...]

U[SE][:pc] dev[:(params)]

U[SE][:pc] dev[:[params]:mnemonicspace]

U[SE][:pc] @usearg

pc	Postconditional expression.
dev	Device name.
param	Device parameter.
params	Device parameters delimited with colon.
mnemonicspace	Expression with routine name.
usearg	Argument indirection.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

The *use* command applies device parameters to specified device and make this device current. Device need to be opened by current process first or be created as principal device on proces start (default device). If process does not open this device first, the *use* command generates <NOTOPEN> error.

All input and output actions done using current device, and to operate several devices need to make appropriate device current before *read* or *write* commands.

In interactive mode (MiniM Console (|CON|), telnet (|TNT|) or standard (|STD|)) MiniM process automatically make principal device current device, using pseudocode

```
use $principal
```

For *use* commands all values *pc*, *dev*, *mnemonicspace* or *usearg* can be specified as expressions. Before execution *use* command this expressions are evaluates iun left-to-right order.

The *use* command can be applied to any already opened device and to current device repeatedly. If *use* command have several arguments delimited by comma, command use specified devices sequentially, in left-to-right order. And after this current device id last device in arguments.

The *mnemonicspace* parameter is a routine name to handle mnemonics. For example:

```
use $io::"COMMON"
```

```
use $io:():"COMMON"
```

Here *use* command assign to *$io* device mnemonic routine ^COMMON and does not change currently used device parameters.

On argument indirection need the *usearg* be a valid *use* argument or arguments. Before execution this expression evaluates as a string and content used as *use* arguments.

All device parameters are device-type specific and listed in special chapter about devices.

After *use* command current device name is in special system variable *$io*. This variable returns current device name without parameters and without mnemonic routine.

3.24 WRITE

Command output data into current device.

Syntax

W[RITE][:pc]

W[RITE][:pc] expr

W[RITE][:pc] *intexpr

W[RITE][:pc] ?intexpr

W[RITE][:pc] writeformat

W[RITE][:pc] /mnemonic[(params)]

W[RITE][:pc] writearg[,writearg2,...]

W[RITE][:pc] @indwritearg

Description

pc	Postconditional expression.
expr	Expression to evaluate and output as string.
intexpr	Expression to evaluate as integer.
writeformat	Formatting symbol.
mnemonic	Mnemonic name.
params	One or more mnemonic parameters.
writearg	One or more possible command arguments.
indwritearg	Argument indirection.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

The *write* command send data into current device and control formatting. The *write* command always use current device only and mnemonic uses only routine assigned as mnemonic routine for current device.

Argumentless *write*

Argumentless *write* command write out all available local variables of current process, which are visible on current stack level and have defined value.

Argumentless *write* command writes out variables in alphabetical sorting order by name and subscripts in index sorting order.

If variable has string value in internal MiniM representation, it outputs as string inside double quotes ("). Otherwise data writes out as number, without quotes.

To make correct syntax for variable values need to be used special system function *$zquote()* to double quotes inside string and add leading and trailing quotes if value is not a number.

Examples:

```
TEMP>s a(1)=1,b(1)="1",b("1 1")="1 1"

TEMP>w
a(1)=1
b(1)="1"
b("1 1")="1 1"

TEMP>
```

Here values of *a(1)* and *b(1)* outputs with different formatting, because internal data representations are different. First case is a number and second case is a string.

Write out string

To write out data as string, command evaluates an expression as string and sends byte sequentially to current device as is. For example:

```
TEMP>w "2*2=",2*2
2*2=4
```

Here *write* command has 2 arguments, first evaluates as string and command output "2*2=", and second is expression, command evaluates as string and output result of 2*2.

Write out symbol of code

The *write* command accept special symbol * to specifi next is expression to evaluate as a number and count as symbol's code. It number need to be integer between 0 and 255 generally to specify byte code to output. Format conversion is like *$char()* function with one argument:

```
TEMP>w *65
A
TEMP>w $c(65)
A
```

The second case equals to first case by byte sequence to output, but differs from first case in *$X* and *$Y* recalculations. In the first case the *write* command does not recalculate *$X* and *$Y* values and in second do.

Tabbed output

Tabbed oitput is specified by special symbol "?" with following expression, which evaluates as integer. It is defined position in line to shift to. For example, this is supported by MiniM console and telnet devices. Other devices support tabbed output dependent of device type. The tabbing expression evaluates only as integer and specifies the position in line and device need to position current output pointer. Depending of device type it can be spaces or device-special positioning. MiniM counts current position in line (as it is possible) using special system variables *$X* and *$Y*.

If evaluated tabstop expression is less then or equal zero, tabbing does nothing. Otherwize device outputs appropriate spaces to shift to specified position. The position can be over the line size, and output continues from the next line. Next output continues from resulting position. For example:

Examples:

```
TEMP>w ?2,"*",?10,"*"
 *       *
TEMP>s tab1=3,tab2=10
```

```
TEMP>w ?tab1,"*",?tab2,"*"
  *      *
```

Here *write* command make tabbed output to position 2, writes symbol "*", next shift position to 10, and symbol "*" again. In the second case tabbed positions are specified by expressions. All expressions used in arguments are evaluates in left-to-right order.

Formatting

To format output there can be used following symbols:

#	Clear the screen, flush buffers, form feed, or other device-specific action to make the similar output clearing.
!	Output the line feed symbol.

For text mode devices MiniM outputs bytes $c(13),$c(10), and for binary mode devices outputs $c(10) only.

Formatting symbols can be delimited by comma, but it is not mandatory, For example:

```
write #!!!
```

here first clears the screen, reposition caret to first line and outputs three empty lines.

Mnemonic usage

If mnemonis is specified, the name should be used as is, without expression or indirection. Tha values passed to mnemonic, are passed by value only, not by reference. The mnemonic name need to be preceded by slash "/". For example:

```
write /CUP(10,20)
```

Here command call mnemonic with name CUP in current mnemonic handling routine and pass two parameters.

If device has not assigned mnemonic routine, mnamonic calling generates the <MNEMSPACE> error, MDC code 26. If program call mnemonic not supported by mnemonic routine, it generates <NOLINE> error.

Argument indirection

The *write* command supports argument indirection. The *indwritearg* is evaluated as string and used as *write* arguments. Empty string does not allowed. Argument indirection can be used with other available argument forms with comma delimiting. For example:

```
TEMP>write #,/CUP(10,20),@line,!

USER>f expr="2*2,!","5*5,!","6*6,!" w expr," : ",@expr
2*2,! : 4
5*5,! : 25
6*6,! : 36
```

3.25 XECUTE

Executes a string as a commands line.

Syntax

X[ECUTE][:pc1] expr[:pc2][,expr2,...]

X[ECUTE][:pc] @xarg

Description

pc	Postconditional expression.
expr	Expression to evaluate and execute as commands.
xarg	Argument indirection.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

The *xecute* command evaluate expression *expr* and use content as command sequence. The *expr* evaluates after evaluating appropriate postconditional expression if one specified.

The *expr* content can have no any commands and containt comment.

If *xecute* command execute argument, it is created new stack level. After full command line execution this stack level leaves. If on this stack level has

been created some local variable using *new* command, they are destroyed.

The *xecute* command is much similar like *do* execution, for subroutines in one command line. The system function *$stack* show this execution context is from command *xecute*, not from *do* command. For example:

```
TEMP>x "n a s a=1"

TEMP>w

TEMP>
```

If postconditional expression *pc1* present, it evaluates as integer and compare with 0. If it is not 0, the *xecute* command applies to all available arguments delimited by comma. If postconditional expression *pc2* is present, it evaluates as integer and compares with 0. If it is 0, command does not use this argument and continue analize next argument. For example:

```
TEMP>x "w 1":1
1
TEMP>x "w 1":0

TEMP>
```

On *xecute* command execution the system variable *$test* does not stacked. If it is changed, it changes on the previous stack level. See for details *$test* documentation.

Chapter 4

Z - Commands

4.1 ZNEW

The *znew* command makes new place of specified local variable on the current stack level with copying all available data from variable with same name including subscripts.

Syntax

ZNEW[:pc]

ZNEW[:pc] lvn[,lvn2,...]

ZNEW[:pc] (lvn[,lvn2,...])

ZNEW[:pc] @argindir

Description

lvn	Local variable name to make local copy.
pc	Postconditional expression.
argindir	Argument indirection

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

The *znew* command is the same as *new* command except it stores in created place all existent data of variable including subscripts. The *new*

command has result only place created, all data undefined.

Argumentless *znew* command applies to all visible on the current stack level local variables. Argumentless *znew* command does not affect to nonexistent names unlike *new* command.

Exclusive form of *znew* command applies to all visible on the current stack level local variables except listed in argument. Exclusive form of *znew* command does not affect to nonexistent names unlike *new* command.

Examples:

```
USER>s a=0,a(1)=1 x "znew  w"
a=0
a(1)=1

USER>s a=0,a(1)=1 x "znew  s a(2)=2 w" w
a=0
a(1)=1
a(2)=2
a=0
a(1)=1

USER>s a=0,a(1)=1,b="b" x "znew  s a(2)=2 w" w
a=0
a(1)=1
a(2)=2
b="b"
a=0
a(1)=1
b="b"

USER>s a=0,a(1)=1,b="b" x "znew  s a(2)=2 s b=""c"" w" w
a=0
a(1)=1
a(2)=2
b="c"
a=0
a(1)=1
b="b"

USER>s a=0,a(1)=1,b="b" x "znew a s a(2)=2 s b=""c"" w" w
a=0
```

```
a(1)=1
a(2)=2
b="c"
a=0
a(1)=1
b="c"

USER>s a=1,b=2 x "znew (a) s a=3,b=4,c=5 w  w !" w
a=3
b=4
c=5

a=3
b=2
c=5
```

The *znew* command is not a part od MUMPS standard, it need to take a proper use to write portable programs. Some other MUMPS implementations can support *znew* command with the same syntax and behavior.

Command can use only unsubscripted local variables names.

If specified argument indirection, expression of *argindir* is evaluated as string and content need to be valid *znew* argument[s], listed by comma.

4.2 ZNSPACE

Command switch current database to specified.

Syntax

ZN[SPACE][:pc] expr[,expr]

ZN[SPACE][:pc] @argindir

Description

expr	Expression with database name to switch to.
pc	Postconditional expression.
argindir	Argument indirection.

Command make specified database current database. Database name evaluates as an expression first. The database name is case-insencitive. If

this database name does not defined for current MiniM instance, process generates <NAMESPACE> error, MDC code 26 (call to nonexisting environment).

```
TEMP>zn "not exist "

<NAMESPACE>
TEMP>w $ec
,M26,
TEMP>
```

If current database is the same to switch to, the *znspace* command does nothing. Side effect to switch database is naked indicator clearing.

```
USER>w $d(^a)
10
USER>w $zr
^a
USER>zn "%sys"

%SYS>w $zr

%SYS>

USER>znspace @"""%sys"""

%SYS>
```

Current database name is returned by special system variable *$znspace*.

```
TEMP>w $znspace
TEMP
TEMP>
```

4.3 ZPRINT

Command outputs lines of routine to the current device as specified in command's argument.

Syntax

ZP[RINT][:pc]

ZP[RINT][:pc] [label][+offset][^[database]routine]

ZP[RINT][:pc] [label1][+offset1][^[database]routine]:[label2][+offset2]

ZP[RINT][:pc] @argindir

Definition

label	Optional label from where need to be counted lines of routine.
offset	Offset from label.
database	Database name from where need to be read routine content.
routine	Name of routine to output.
argindir	Value with argument, indirection.

Command ZPRINT may be used in the following forms:

1. In argumentless form command outputs to the current device all content of the current routine.
2. If command got only routine name, command outputs all content of the routine specified.
3. If first label is present, command outputs from the specified label and only one line.
4. If both parts are present, command outputs lines from the first line to the second line including ones.

Before writing content command check that both labels exists in the routine. If one of line (first or second) does not contains in the routine, or second line corresponds before first line, command does nothing.

Command accepts first label in the same format as function $TEXT, and any part of argument may be specified using indirection. Any part of first and second part may be omitted, and command counts lines using only available specification. If argument does not contains routine name, command uses currently executed routine. If database name is omitted, command uses current database name.

```
USER>zp
```

Command does nothing becouse top-level of interactive device does not have current routine.

```
USER>zp ^uuuunnnn
```

Command does nothing becouse argument specify unexisting routine name.

```
USER>zp ^%DBCRC
%DBCRC ; MiniM system utilities, Check database CRC
 n ans,list,i,dbname,err
 s list=$v("db",15)
 w !,"MiniM database CRC check utility",!
start
 w !,"Available database list:",!
 f i=1:1:$l(list,"*") w " ",i,") ",$p(list,"*",i)
 r !,"Select database number to check: ",ans,!
 i '+ans q
 s dbname=$p(list,"*",ans)
 i dbname="" q
 s err=$v("db",21,dbname)
 i err=0 w "Database ",dbname," CRC check database defect.",!
 e  w "Database ",dbname," CRC check OK.",!
 g start

USER>
```

Command outputs entire routine content becouse command argument does not contains first and second line limits.

```
USER>zp +3^%DBCRC
 s list=$v("db",15)
```

Command outputs only the one specified line.

```
USER>zp +2^%DBCRC:+4
 n ans,list,i,dbname,err
 s list=$v("db",15)
 w !,"MiniM database CRC check utility",!
```

Command writes lines of routine from first to second including ones and uses offsets.

```
USER>zp start^%DBCRC:start+4
start
 w !,"Available database list:",!
 f i=1:1:$l(list,"*") w " ",i,") ",$p(list,"*",i)
 r !,"Select database number to check: ",ans,!
 i '+ans q
```

Command writes lines of routine from first to second including ones and uses labels.

It is much important to know that the second part of argument does not use routine name. Command uses name of routine from the first part of argument or uses currently executed routine.

Command ZPRINT use only INT routines and does not use MAC or INC routine with the same name. If database contains compiled bytecode and does not contain source code of INT routine, command does nothing.

4.4 ZSYNC

Command activate a write and journal daemons.

Syntax

ZSYNC[:pc]

Description

pc	Postconditional expression.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

The *zsync* command send signals to write and journal daemons to flush changed cache blocks and available journal buffer independently of current internal daemon conditions. Control returns immediately after signals sent.

Daemons got signal and execute write changed cache blocks and journal buffer.

Daemons write changed data for all MiniM instrance, not only for signaled process.

If control return from *zsync* command, this does not mean flush is done, it is only signals are sent.

Command invented to speed up database flushing in special cases.

4.5 ZTRAP

Command generates the default or specified error.

Syntax

ZT[RAP][:pc]

ZT[RAP][:pc] expr

ZT[RAP][:pc] @argindir

Description

expr	Expression to use to make error name.
pc	Postconditional expression.
argindir	Argument infirection.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

Argumentless form of *ztrap* command generates the <ZTRAP> error.

Argumented form of *ztrap* command generates error with text specified in argument. The *expr* is evaluated as a string and command get forst up to 32 symbols and constructs error name as symbol Z and first 32 symbols from *expr* value. Command adds to current *$ecode* value string ZZTRAP.

Examples:

```
USER>ztrap

<ZTRAP>
```

```
USER>w $ze
<ZTRAP>
USER>w $ec
,ZZTRAP,
USER>ztrap "MY ERROR"

<ZMY ERROR>
USER>w $ec
,ZZTRAP,ZZTRAP,
USER>w $ze
<ZMY ERROR>
USER>s errcode="Failed"
USER>ztrap @"errcode"

<ZFailed>
```

4.6 ZWRITE

Command writes to current device all visible local variables or specified local or global variable with subscripts.

Syntax

ZW[RITE][:pc]

ZW[RITE][:pc] glvn[,glvn2,...]

ZW[RITE][:pc] @argindir

Description

glvn	Local or global variable name to write out.
pc	Postconditional expression
argindir	Argument indirection.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

Argumentless form of the *zwrite* command write to current device all

visible local variables with subscripts.

Examples:

```
TEMP>k

TEMP>s a="a",a(1)="a1",b="b",b(1)="b1"

TEMP>zw
a="a"
a(1)="a1"
b="b"
b(1)="b1"

TEMP>

TEMP>zw @"a,b"
a="a"
a(1)="a1"
b="b"
b(1)="b1"
```

Argumented form of *zwrite* command write to current device specified local or global variable with subscripts.

```
TEMP>zw a
a="a"
a(1)="a1"

TEMP>zw a(1)
a(1)="a1"

TEMP>
```

In the case of specified variable does not have value or any subscripts, the *zwrite* command does nothing with this variable and does not generate <UNDEFINED> error.

```
TEMP>k
```

```
TEMP>s a(1)=1

TEMP>zw
a(1)=1

TEMP>zw a
a(1)=1

TEMP>
```

The *zwrite* command outputs subscripts in index sorting order.

If command have several arguments, command write all specified variables in left-to-right order.

```
TEMP>k

TEMP>s a="a",b="b",c="c",d="d"

TEMP>zw a,c
a="a"
c="c"

TEMP>
```

4.7 ZZDUMP

Command write to current device hexadecimal dump of string's expression representation.

Syntax

ZZDUMP[:pc] expr[,expr2,...]

ZZDUMP[:pc] @argindir

Description

expr	Expression to evaluate as string and to dump.
pc	Postconditional expression.
argindir	Argument indirection.

Postconditional expression evaluates before command executes and result compares with 0. If expression is not 0, command executes. If expression is 0, command is skipped and executes next command in string. If specified command with several arguments delimited with comma, postconditional expression applied to all arguments. Command applies to argument in left-to-right sequence as specified.

Command outputs to current device the hexadecimal dump of specified expression value. It is 3 columns: 1) segment number (counts from 0000), 2) hexadecimal byte codes and 3) printable byte's representation. If byte is not printable, it represented with dot.

```
USER>zzdump $c(0,1,2,3,4,35,36,37,38,46,
       47,48,78,79,80)
0000: 00 01 02 03 04 23 24 25 26 2E 2F
       30 4E 4F 50           .....#$%&./0NOP
```

If specified several expression comma delimited, command evaluates and dump specified expresions in left-to-right order.

```
TEMP>zzdump 123,456
0000: 31 32 33                 123
0000: 34 35 36                 456

USER>s a=1,b=3

USER>zzdump @"a,b"
0000: 31                       1
0000: 33                       3
```

If argument evaluates as an empty string, the *zzdump* command does nothing.

Chapter 5

Standard Functions

5.1 $ASCII

Return decimal value of specified ASCII symbol.

Syntax

$A[SCII](String{,Position})

Definition

String	Any string expression.
Position	Integer expression, position of symbol in *String*.

The $ASCII function return integer value as decimal code of ASCII symbol within specified string and position. Decimal value counts as an ASCII code of symbol of *String* in position *Position*. If *Position* argument does not specified, it is used as 1. Function return result as -1 for an empty string, or if *Position* is less then 1 or outside the *String* range.

Examples:

Command	**Function result**
`S X="ABCDE"`	$A(X)=65
	$A(X,1)=65
	$A(X,2)=66
	$A(X,3)=67
`S Y="4"`	$A(X,Y)=68
`S X=""`	$A(X)=-1
	$A(X,n)=-1 with any n.

```
S X="AB"
```

$A(X,0)=-1
$A(X,3)=-1
$A(X,-7)=-1
$A(X,1.92)=65 used integer expression part.

5.2 $BIT

Return bit value (0 or 1) from specified bitstring in specified position.

Syntax

$BIT(bitstring,position)

Definition

bitstring	Bitstring created by *$bit* functions.
position	Integer expression, bit position.

Function *$bit()* return bit value from bitstring in specified position. If this string have not bitstring format, function generates an error <INVALID BIT STRING>. If value of *position* expression is outside of possible range, function generates an error <RANGE>. Range allowable is from 1 to 262104 inclusively.

If instead bitstring is used empty string or vvariable with undefined value, function always return 0 value. If *position* is outside of available bits in *bitstring*, function return 0 value.

Examples:

```
USER>s $bit(a,3)=1,$bit(a,6)=1

USER>f i=1:1:10 w $bit(a,i)
0010010000
```

5.3 $BITCOUNT

Returns count of available bits in bitstring.

Syntax

$BITCOUNT(bitstring[,bitvalue])

Definition

bitstring	Bitstring in $bitXXX functions format.
bitvalue	Bits value to count (0 or 1).

Function *$bitcount()* with one argument return total bits count available in *bitstring*, and with two argument returns count of specified bits (0 or 1). In two-argument form the *bitvalue* expression evaluates as integer and compared with 0. If it is 0, function calculate total count of 0 bits, otherwise calculate total count of 1 bits.

If instead of *bitstring* was specified variable with undefined value or expression with empty string value, function returns 0.

If the value of *bitstring* has invalid bitstring format, function generates an error <INVALID BIT STRING>.

Examples:

```
USER>s $bit(a,3)=1,$bit(a,12)=1

USER>w $bitcount(a)
12
USER>w $bitcount(a,1)
2
USER>w $bitcount(a,0)
10
USER>w $bitcount(aaaaa)
0
USER>w $bitcount(123)

<INVALID BIT STRING>
```

5.4 $BITFIND

Return position of bit with specified value.

Syntax

$BITFIND(bitstring,bitvalue[,position[,direction]])

Definition

bitstring	Bitstring to find bit in.
bitvalue	Bit value to find in *bitstring*.
position	If specified, is a start bit position to find from.
direction	If specified, is a find direction - forward or backward.

Function *$bitfind()* returns position of specified bit in *bitstring*. If next bit position not found, function returns value 0.

If the *bitstring* is an undefined variable or expression with an empty string, function return value 0.

If value of *bitstring* has illegal format, function generates an error <INVALID BIT STRING>.

The *bitvalue* expression evaluates as an integer and compares with 0. If it is 0, function search next bit with 0 value, otherwise search next bit with 1 value.

Function counts bits position from 1, and starting bit position is 1.

If argument *position* is specified, search begins from this position. If specified *position* less than 1, search begins from first position. If argument *position* is not specified (2-argument form), search begins from first position.

If argument *direction* is specified, this expression evaluates as an integer and compares with 1 and -1. If it is 1, search starts forward, id -1, search starts backward, otherwise function generate an error <FUNCTION>. If value of *direction* is not specified (2- or 3-argument forms), search starts forward.

Examples:

```
USER>s $bit(a,3)=1,$bit(a,5)=1

USER>s i=0 f  s i=$bitfind(a,1,i+1) q:'i  w i,!
3
5

USER>s i=0 f  s i=$bitfind(a,0,i+1) q:'i  w i,!
1
2
4
```

Here code creates bitstring with bit 1 in positions 3 and 5, and display 0 and 1 bits positions is available in bitstring.

5.5 $BITLOGIC

Evaluates bitwise operation AND, OR, NOT over bitstrings and returns result as new bitstring.

Syntax

$BITLOGIC(bitexpression)

Definition

bitexpression	Logical bitwise expression over bitstrings with AND, OR and NOT operations.

Function *$bitlogic()* evaluates bitwise AND, OR and NOT operations over bitstring. Syntax rules for the *bitexpression* are:

```
bitexpression = bitatom [ bittail ]

bitatom = |      glvn       |
          | (bitexpression) |
          |    ~bitatom     |

bittail = bitoper bitatom

bitoper = | & |
          | | |
```

Here *glvn* is a local or global variable name.

Expression - argument of function *$bitlogic()* evaluates in left-to-right order if order does not specified by parenthesis. Math priorities of operations from boolean algebra does not applied.

Operator NOT ( ) is defined as unary operator and applied to right operand. Operators AND (&) and OR (—) are defined as binary operators and applies to left and right operands.

Instead bitstrings van be used local or global variable names or empty strings, and for empty strings bitstrings counts as bitstrings from zero only bits. The NOT operation result has the same bits count as present in argument. And AND and OR operation result has a maximum bit count of left and right operands. If one of operands of binary operators is shorter, it logically appends by zero bits to fit need length.

If one of logical operands has illegal bitstring format, function generates an error <INVALID BIT STRING>.

Examples:

```
USER>s $bit(a,1)=1,$bit(a,3)=1,$bit(a,5)=1

USER>s $bit(b,2)=1,$bit(b,3)=1,$bit(b,4)=1

USER>s c=$bitlogic(~a)

USER>f i=1:1:$bitcount(c) w $bit(c,i)
01010
USER>s c=$bitlogic(a&b)

USER>f i=1:1:$bitcount(c) w $bit(c,i)
00100
USER>s c=$bitlogic(a|b)

USER>f i=1:1:$bitcount(c) w $bit(c,i)
11111
```

5.6 $CHAR

Function return a string of symbols with specified decimal ASCII codes.

Syntax

$C[HAR](Integer{,...})

Definition

Integer	Expression to count as a decimal ASCII code.

Function *$CHAR* return a string of symbols with specified ASCII codes and with length of non-negative arguments count. If value of *Integer* is less than 0 or greater than 255, function skip this position or make an empty string. Maximum arguments count supported is 255.

Examples:

Command	**Function result**

`S X=65,Y=66,Z="GOB"`	$C(X)=”A” $C(Y)=”B” $C(X,Y)=”AB” $C(X,Y,67)=”ABC” $C(X,-1,Y)=”AB” $C(-1)=”” (empty string) $C(65.7)=”A” (Used integer part of expression) $C($A(Z,1),$A(Z,2),$A(Z,3))=”GOB”

5.7 $DATA

Returns indicator contain variable value and defined subscripts or not.

Syntax

$D[ATA](lvn[,glvn])

$D[ATA](gvn[,glvn])

$D[ATA](ssvn[,glvn])

Definition

lvn, gvn, ssvn, glvn	Local, global or structured system variable need to check have data.

Function *$DATA()* return indicator as an integer. Function can return of the following values:

0	Variable have no data and subscripts.
1	Variable have data but no subscripts.
10	Variable have no data but have subscripts.
11	Variable have data and subscripts.

```
TEMP>k  s a="",b(1)="",c="",c(1)=""

TEMP>w
a=""
b(1)=""
c=""
c(1)=""
```

```
TEMP>w $d(z),!,$d(a),!,$d(b),!,$d(c)
0
1
10
11
```

Global variable name can be specified as name, name with subscripts, full name with extending syntax with database name and using naked indicator.

Function *$data()* with global variable name always have a side effect - naked indicator is changed even if this global name does not have any data and subscripts defined.

```
TEMP>k ^a,^b

TEMP>s ^a=1

TEMP>w $d(^a),!,$zr,!,$d(^b),!,$zr
1
^a
0
^b
```

Function *$data()* with structured system variable return information about existence of this subscript and possibility and result is dependent of this ssvn type.

Function *$data()* with ^$LOCK check existence of specified in first subscript lock. This lock can be made by this process as such as other process.

```
TEMP>l  w $d(^$L("a(1)")),! l a(1) w $d(^$L("a(1)"))
0
1
```

Function *$data()* with ^$JOB check existence of specified in first subscript job number.

```
TEMP>w $d(^$J(123)),!,$d(^$J($j))
0
1
```

Function *$data()* with ^$ROUTINE check existence of specified in first subscript routine's source code.

Function *$data()* with ^$GLOBAL check existence of specified in first subscript global name. Global exists if any subscript of global or unsubscripted name have data.

```
TEMP>w $d(^a)
1
TEMP>w $d(^$G("^a"))
1
```

Function *$data()* with ^$DEVICE check opened or not device specified in first subscript. All devices in MiniM Database Server are process-private only and ^$DEVICE return information only about devices which are opened by current process.

```
TEMP>w $d(^$D($io)),!,$d(^$D("fgbfdb gfbfdb fgb"))
1
0
```

If two-argument form of the *$data()* function is used, second argument must be the local or global variable name, and in the case of specified in the first argument variable has the value, this value is assigned to the variable *glvn*.

5.8 $EXTRACT

Return substring of string.

Syntax

$E[XTRACT](*String*[,*Start*[,*End*]])

Definition

String	Source string to extract from.
Start	Position to extract from.
End	Position to extract to.

Function *$extract()* return substring of *String* from *Start* position to *End*

position inclusively.

If source *String* is an empty string, function always return empty string. If *Start* position does not specified, it supposed 1 (string begin). If *End* does not specified, it supposed equal *Start* position and function return only one symbol. If *Start* position is greater than *String* length, function return an empty string. If *Start* position is less than 1, function return result from *String* begin. If *End* is greater than *String* length, function return substring up to end of *String*. And, if *End* is less than *Start*, function always return an empty string.

Examples:

```
>w $e("abcd")
a
>w $e("abcd",3)
c
>w $e("abcd",3,6)
cd
```

5.9 $FIND

Function search substring in string and return position of next symbol after found substring entry.

Syntax

$F[IND](String,Substring[,Start])

Definition

String	Expression evaluated as a string to search *Substring* entry.
Substring	Expression evaluated as a string to search in *String*.
Start	Optional integer expression, position to start search *Substring* in *String*. If omitted, function search from begin of *String*.

Function *$find()* search *Substring* in *String* and return position after found entry of *Substring*.

```
TEMP>w $f("123456789",23)
4
```

If function does not found *Substring* in *String*, it return 0.

```
TEMP>w $f("123456","789")
0
```

Optional argument *Start* specify position to start search from this position.

```
TEMP>w $f("123123","3",2)
4
TEMP>w $f("123123","3",4)
7
```

So, function *$find()* can return position which does not belong any symbol inside *String*.

If value of *String* evaluates as an empty string, function always return an empty string and does not use any other arguments, but arguments evaluates in left-to-right order.

```
TEMP>w $f("",123)
0
```

If value of *Substring* evaluates as an empty string, function always return the *Start* position.

```
TEMP>w $f("123","")
1
TEMP>w $f("123","",456)
456
```

5.10 $FNUMBER

Return formatted numeric value.

Syntax

$FN[UMBER](Number,Format{,Fraction})

Definition

Number	Expression evaluates as a number to format.
Format	Formatting specification.
Fraction	Number of digits after decimal point, optional.

Function *$FNUMBER* formats numeric value using *Format* specification and optional specification digits after decimal point.

Format **specification for $FNUMBER function**

Code	**Function result**
`+ (plus)`	Adds plus symbol if *Number* has positive value.
`- (minus)`	Suppress leading minus sign for negative *Number*.
`, (comma)`	Insert comma symbol each 3 digits left of decimal point to separate thousands except leading symbol.
`. (dot)`	Insert dot symbol each 3 digits left of decimal point to separate thousands except leading symbol and uses comma instead of decimal point.
`T or t`	Formats *Number* with plus or munus sign after *Number* or add one more space if minus suppressed by "-" format specifier.
`P or p`	Formats negative *Number* in parenthesis or positive *Number* between spaces.

Format specification symbols can be used in any order. If one or more specifiers present more than once, second entry is ignored. If *Format* expression evaluates as an empty string, function returns source value of *Number*. If *Format* contains both "P" and "T" or both "+" and "-", function generate an error.

The *Fraction* argument evaluates as an integer and used as a fractional digits count to format *Number*. Function $FNUMBER formats *Number* with rounding and add if need trailing zeroes.

Examples:

Command	**Function result**

```
SET X=3.14159           $FNUMBER(X,"+")="+3.14159"
                        $FN(X,"+T")="3.14159+"
                        $FN(X,"+T",4)="3.1416+"
                        $FN(X,"T",4)="3.1416 "
                        $FN(X,"P",6)=" 3.141590 "
                        $FN(X,"P",5)=" 3.14159 "
                        $FN(X,"P",4)=" 3.1416 "
                        $FN(X,"P",3)=" 3.142 "
                        $FN(X,"P",2)=" 3.14 "
                        $FN(X,"P",1)=" 3.1 "
                        $FN(X,"P",0)=" 3 "
S X=1234567             $FN(X,",",2)="1,234,567.00"
S X=-15.406             $FN(X,"T")="15.406-"
                        $FN(X,"T",2)="15.41-"
                        $FN(X,"P")="(15.406)"
                        $FN(X,"PT-") error <FUNCTION>
```

5.11 $GET

Return value of specified variable if it exists.

Syntax

$G[ET](name[,default])

Definition

name	Local, global or structured system variable to read if have data.
default	Expression to evaluate and return id variable have not data. By default counts as an empty string.

Function *$get()* with local or global variable return value of specified variable name if this name have data. Otherwise function return value of *default* expression.

```
TEMP>k  w $g(a,"a"),!,$g(a(1),"1")
a
1
```

Function *$get()* have a side effect - if function call global variable, it changes a naked indicator value.

```
TEMP>k ^a w $g(^a,"a"),!,$zr,!,$g(^a(1),"a1"),!,$zr
a
^a
a1
^a(1)
```

Function *$get()* with structured system variable have result depended of this variable meaning. For example, ^$LOCK variable return locking owner and locking count.

```
TEMP>w $g(^$J($j))

<SSVN VALUE>
TEMP>w $g(^$D($io))

<SSVN VALUE>
TEMP>l a(1) w $g(^$L("a(1)"))
1520:1
```

5.12 $JUSTIFY

Return string right-aligned with specified width and decimals.

Syntax

$J[USTIFY](String,Width[,Decimals])

Definition

String	String to format.
Width	Width to align.
Decimals	Decimal digits after decimal dot.

Function *$justify* formats value of *String* right-aligned within the specified *Width* and add spaces left as need. If argument *Decimals* is specified, value of *String* formats as number with specified number of digits after decimal point and round as need.

If argument *Decimals* is not specified, function does not round value of *String*. To round value of *String* to integer without fractional value of *Decimals* need to be 0.

If value of *Decimals* is less than 0, function rounds value of *String* to integer.

If *Width* is less than *String* length, function return value of *String* without truncation. If *Decimals* specify result string need to be longer than *String* length, funciton does not truncate result.

Examples:

```
>w $j("12.27qwer",12)
   12.24qwer
>w $j("12.27qwer",12,1)
   12.3
>w $j(-"12.27qwer",4,8)
-12.27000000
```

5.13 $INCREMENT

Increments value of specified local or global variable atomic and return result.

Syntax

$INCREMENT(name)

$INCREMENT(name,add)

Definition

name	Local or global variable name to increment.
add	If specified, used as an increment value to add. By default is 1.

Function *$increment()* evaluates value of specified name as a number, adds value of *add* expression (1 by default), save and return result.

If *name* variable has not data, function counts it as a zero.

If *name* variable has nonnumeric data, function counts it as a zero.

If *name* variable is a global variable, function perform unrollable by *trollback* command changes.

Instead of *add* expression can be used any expression, and function evaluates it as a number. Result can be integer or fractional, positive or negative.

The *trollback* command rolls back current transaction by journal in back order. And, if the same variable has been changed by *set* command first and

by *$increment()* function next, *trollback* command does not use *$increment()* changes made, but rolls back changes mede by *set* command.

Examples:

```
USER>k ^a s ^a=123 ts  s ^a=^a+1 w ^a,! tro  w ^a,!
124
123

USER>k ^a s ^a=123 ts  w $i(^a),! tro  w ^a,!
124
124

USER>k ^a s ^a=123 ts  s ^a=456 w $i(^a),! tro  w ^a,!
457
123
```

Function *$increment()* executes atomic, does not lock name used and ignore any lock made by other processes.

Function *$increment()* does not included into current MUMPS Standard.

5.14 $LENGTH

Return string length or count of entries of substring.

Syntax

$L[ENGTH](String[,Substring])

Definition

String	Expression evaluated as string to count.
Substring	Expression evaluated as string to specify substrings in *String*.

Function *$length()* returns number of symbols in *String* id optional argument *Substring* does not specified. If argument *Substring* is specified, function counts number of entries of *Substring* in the *String* and returns result plus one. If value of *String* is empty string, function always return 0.

Examples:

Command	Function result
SET X="ABC"	$LENGTH(X)=3
S X="123456789"	$L(X)=9
S X=""	$L(X)=0
S X="AAAA"	$L(X,"AA")=3
S X="ABCDBCABCABCD"	$L(X,"AB")=4
	$L(X,"DC")=1
	$L(X,"ABCD")=3
	$L(X,"")=0

5.15 $LIST

Return value of list element or sublist of list.

Syntax

$LIST(list[,pos[,end]])

$LI(list[,pos[,end]])

Definition

list	Expression with *$listbuild()* list format.
pos	Position of list element or sublist.
end	Position of last sublist element.

Function *$list()* return element of list in specified position or sublist of list.

If *pos* and *end* arguments does not specified, function return first element of list.

If *pos* argument is specified but *end* does not, function return list element from specified position. If value of *pos* evaluates as -1 integer, function return value of last list element. If value of *pos* evaluates as 0 or 1, function return value of first list element. If value of *pos* evaluates as an integer less than -1, function generates an error <RANGE>, it is range violation. Otherwise function return value of list element at *pos* position.

If value of *list* contain list element but this element have no data, function generate an error <NULL VALUE>. If specified position does not contain any list element (position outside of list), function generate an error <NULL VALUE>.

If function detect that value of *list* have invalif list format, function generate an error <LIST>.

Examples:

```
TEMP>s list=$lb("a","b","c")

TEMP>w $li(list,-1)
c
TEMP>w $li(list,-2)

<RANGE>
TEMP>w $li(list,1)
a
TEMP>w $li(list,2)
b
TEMP>w $li(list,3)
c
TEMP>w $li(list,4)

<NULL VALUE>
TEMP>w $li(123456,3)

<LIST>
```

If specified both *pos* and *end* arguments, function return not list element, but sublist with list structure. New sublist is part of list from *pos* to *end* positions.

If *pos* or *end* evaliates as negative integers, function *$list()* generates an error <RANGE>.

If value of *end* evaluates less than or equal *pos*, function return list with one only element.

If *pos* or *end* arguments are positions outside of real list elements, function *list* does not add empty list elements, and result does not contain data what is absent in the source *list*.

Examples:

```
TEMP>s list=$lb("a","b","c")

TEMP>w $li(list,1,2)
```

```
??a??b
TEMP>w $li(list,2,2)
??b
TEMP>w $li(list,2,3)
??b??c
TEMP>w $li(list,2,8)
??b??c
```

Here symbols ? emphasize nonprintable symbols.

Function *$list()* use an empty string as a valid list structure with no any list elements. 1- and 2- argument forms of *$list()* always generate an error <NULL VALUE> and 3-argument form of *$list()* always return empty string.

Function *$list()* and internals of structures does not included into current MUMPS Standard.

5.16 $LISTBUILD

Return string encoded as list of specified elements.

Syntax

$LISTBUILD([item[,item...]])

$LB([item[,item...]])

Definition

item	List element in appropriate position.

Function *$listbuild()* return string encoded as special structure, as list of arguments values. Each function argument is a list element in this position and follows in arguments order.

Result of *$listbuild()* function and this list structure is used by other *$listXXX* functions.

Arguments can be omitted in any position. If argument omitted this list element in result is present bu contain no data. For example, *$listbuild()* function without arguments return list with one element and without data.

For group of functions *$listXXX* an empty string is a valid list structure with no any elements.

If *$listbuild()* function argument is a local or global variable with no data (undefined value), function make this element too without data.

If list element is present but contain no data, function *$list()* for this list position generate error <NULL VALUE>, and function *$listdata()* return default value.

Any argument of *$listbuild()* function can be result of *$listbuild()* function too. This will be list with element as list.

Result of *$listbuild()* encoding have format which allow concatenate two lists to get new list with element of first and following second list elements.

Examples:

```
TEMP>s list=$lb("a",a,)

TEMP>w $li(list,1)
a
TEMP>w $li(list,2)

<NULL VALUE>
TEMP>w $li(list,3)

<NULL VALUE>
TEMP>w $li(list,4)

<NULL VALUE>
```

Here first list element is defined, second is undefined because used undefined variable, third is undefined because is omitted, and fourth is undefined becouse does not exists.

Internal list encoding have not canonic encoding format and lists cannot be compared by comparision operator, it is need to use special function *$listsame()*.

Examples:

```
TEMP>s list1=$lb(123),list2=$lb("123")

TEMP>w list1=list2
0
TEMP>w $ls(list1,list2)
1
```

Lists encoding format use much shorter numbers encoding if possible and differs from string representation of the same value.

Function *$listbuild()* and internals of list structures does not included into current MUMPS Standard.

5.17 $LISTDATA

Return indicator is defined list element or not.

Syntax

$LISTDATA(list,[pos])

$LD(list,[pos])

Definition

list	Expression with list structure.
pos	Position of element to check.

Function *$listdata()* check and return information about contain data specified list element or not. If *pos* argument is present, this expression evaluates as integer and used as position number. If value of *list* expression is not valid list structure, function generate an error <LIST>. If value of *pos* is less than -1, function generates an error <RANGE>. If value of *pos* is equal -1, function check data for last available list element. For position 0 or over available list positions function always return value 0. Otherwise function return 0 if list element have no data or 1 if have.

If argument *pos* does not specified, function check defined or not first list element.

Note, function does not check list element existence, function check have list element data or not.

Examples:

```
TEMP>s list=$lb("a",a,)

TEMP>w $ld(list,-3)

<RANGE>
TEMP>w $ld(list,-1)
```

```
0
TEMP>w $ld(list,0)
0
TEMP>w $ld(list,1)
1
TEMP>w $ld(list,2)
0
TEMP>w $ld(list,3)
0
TEMP>w $ld(list,4)
0
```

Here are different cases to check list element have data or not. If position is less than -1, function generate an error <RANGE>, for 0 position list element have no data, in position 1 data exists, in position 2 element have no data because was used undefined variable to build list, 3 element have no data because *$listbuild()* element was omitted, and 4 position is absent.

Function *$listdata()* and internals of list structures does not included into current MUMPS Standard.

5.18 $LISTFIND

Function search data in list.

Syntax

$LISTFIND(list,value[,startafter])

$LF(list,value[,startafter])

Definition

list	Expression with list value.
value	Search value to find as list element.
startafter	Position to start search.

Function *$listfind()* search in list elements entry with specified value.

If function find list element with specified value, funciton return list position of this element, and position counts from 1. If not found any element, function return value 0. If list contains other lists as elements, function does not search recursively. For each list element function use the "equals" oper-

ator to compare values, and logically equal lists can be not found, because list structure has not canonical data represendation. For example:

```
TEMP>w $lf($lb($lb("1")),$lb(1))
0
```

Here value does not found because internal in-byte representation of *$lb(1)* and *$lb("1")* are different.

Optional third argument specify start position to search list element after this position and this position does not used to search. Argument *startafter* evaluates as an integer and if value is less than or equal 0, function search from first list element.

If *startafter* argument specifies list position after last available, function return value 0.

Example how value can be found in list:

```
TEMP>s list=$lb("a","b","c","a","d")

TEMP>s p=0 f  s p=$lf(list,"a",p) q:p=0  w p,!
1
4

TEMP>
```

Here we create list where value "a" is present in 1 and 4 position. Variable *p* used as list position to find element after this. Code use cycle which ends if function *$listfind()* returns value 0. And code write out found positions.

If function *$listfind()* detect list structure is corrupted, function generate an error <LIST>. For example:

```
TEMP>w $lf(123,123)

<LIST>
```

Function *$listfind()* cannot find in list element with undefined value any data including empty string.

Function *$listfind()* and internals of list structures does not included into current MUMPS Standard.

5.19 $LISTFROMSTRING

Function creates list from the delimited string.

Syntax

$LISTFROMSTRING(string[,delim])

$LFS(string[,delim])

Definition

string	Source string with delimiters, which items need to be used as list items.
delim	Delimiter for resulting string. If this argument is omitted, function uses a comma as a default string delimiter.

Function $listfromstring() uses source delimited string and creates a list with the same items.

If result exceeds limit for one string in bytes, funcion generates an error ¡MAXSTRING¿.

Examples:

```
USER>s str="11,22,33"

USER>s list=$lfs(str)

USER>f i=1:1:$ll(list) w $li(list,i),!
11
22
33

USER>s str="11~22~33"

USER>s list=$lfs(str,"~")

USER>f i=1:1:$ll(list) w $li(list,i),!
11
22
33
```

Developers need to know that, unlike function $listfromstring() is a pair function to the $listtostring(), the result of sequencial appliyng of functions

$listtostring() + $listfromstring() may differ from the source list, becouse list internal structure does not have canonical representation. So, ff developers need to compare list structures, there must not be used binary comparision, and must be used special logical comparision function $listsame().

Example:

```
USER>s list1=$lb(11,22,33)

USER>s str=$lts(list1)

USER>s list2=$lfs(str)

USER>w list1=list2
0
USER>w $listsame(list1,list2)
1
```

Function $listfromstring() and internals of list structures does not included into current MUMPS Standard.

5.20 $LISTGET

Return list element or default value.

Syntax

$LISTGET(list[,pos[,default]])

$LG(list[,pos[,default]])

Definition

list	Expression with list value.
pos	List element position.
default	Default value.

Function *$listget()* return list element value if this element have data or default value if list element have undefined value.

Value of *list* expression need to be valid list structure, created by *$listbuild()* function.

If argument *pos* is omitted, function return value of first list element, from

position 1.

If value of *pos* is less than -1, function generate an error <RANGE>.

If value of *pos* evaluates as -1, function return value of last available list element.

If value of *pos* evaluates as 0 or specify position outside of really present list elements, function return default value.

If argument *default* is omitted, function use an empty string as default value.

If function detect that value of *list* is not valid list structure, function generate an error <LIST>.

Examples:

```
TEMP>s list=$lb("a","b","c")

TEMP>w $lg(list,-3,"def")

<RANGE>
TEMP>w $lg(list,-1,"def")
c
TEMP>w $lg(list,0,"def")
def
TEMP>w $lg(list,1,"def")
a
TEMP>w $lg(list,2,"def")
b
TEMP>w $lg(list,3,"def")
c
TEMP>w $lg(list,4,"def")
def
TEMP>w $lg(123,4,"def")

<LIST>
```

Function *$listget()* and internals of list structures does not included into current MUMPS Standard.

5.21 $LISTLENGTH

Function return count of available list elements.

Syntax

$LISTLENGTH(list)

$LL(list)

Definition

list	Expression with list value.

Function *$listlength()* return count of available list elements of *list*. Value of *list* must have list structure created by *$listbuild()* function.

If function detect that *list* have not valid list structure, function geberate an error <LIST>.

For *list* as an empty string function return value 0.

For other list structures function return count of available list element, including elements with and without data.

If list element of *list* is a list structure too, function count this element as one element and does not parse internal structure.

Examples:

```
TEMP>w $ll("")
0
TEMP>w $ll("1")

<LIST>
TEMP>w $ll($lb("1"))
1
TEMP>w $ll($lb("1",,))
3
```

Function *$listlength()* and internals of list structures does not included into current MUMPS Standard.

5.22 $LISTSAME

Compares two lists and return equal lists or not.

Syntax

$LISTSAME(list1,list2)

$LS(list1,list2)

Definition

list1,list2	Expression with list value.

Function *$listsame()* compares two lists and return 1 if lists are logically equal (literally), otherwize return 0. List items with undefined values are logically equal.

Empty strings function *$listsame()* count as an empty lists with 0 items count.

Function compares list items sequentially in left-to-right order.

If function detects that list items are not equals, function stop scanning and return 0.

If function *$listsame()* detect that one of list have invalid list structure, function stop scanning and generate an error <LIST>.

If list element have list structure too, function *$listsame()* does not compare elements as lists, and compares literally only.

Examples:

```
TEMP>w $ls("","")
1
TEMP>w $ls("","1")
0
TEMP>w $ls("1","")
0
TEMP>w $ls($lb(1),$lb("1"))
1
TEMP>w $lb(1)=$lb("1")
0
```

Function *$listsame()* and internals of list structures does not included into current MUMPS Standard.

5.23 $LISTTOSTRING

Function creates string with delimiters from the list.

Syntax

$LISTTOSTRING(list[,delim])

$LTS(list[,delim])

Definition

list	Expression with source list value, need to be transformed to the delimited string.
delim	Delimiter for resulting string. If this argument is omitted, function uses a comma as a default string delimiter.

Function $listtostring() creates delimited string from the list with delimiter secified in the second argument. Function does not parses embedded strings as possible list structures and uses ones as is.

If source list value does not match list structure, function generates error ¡LIST¿.

If function detect that one or more list elements have an undefined value, function generates error ¡NULL VALUE¿.

Examples:

```
USER>w $lts($lb())

<NULL VALUE>

USER>w $lts($lb("zero","first","second"))
zero,first,second
USER>w $lts($lb("zero","first","second"),"~")
zero~first~second
USER>w $lts("ordinal string","~")

<LIST>
```

Developers need to know that the function $listtostring() uses items of the source list *list* as is, even if ones includes used delimiter, so number of items of delimited string can differ from the number of list items.

Example:

```
USER>w $lts($lb("a,b","c,d"))
a,b,c,d
```

Here has been used list of two items, each of one contains used delimiter (comma) and result contains three delimiter.

Function $listtostring() and internals of list structures does not included into current MUMPS Standard.

5.24 $LISTVALID

Function check that the argument evaluates as a possible list structure.

Syntax

$LISTVALID(expr)

$LV(expr)

Definition

expr	Expression with possible list value, need to be checked.

Function $listvalid() check argument, is it correct list structure or not. *expr* value can be created by the $listbuild() function or concatenation or any other operation and can conform to the list structure or not.

If argument can be used as a correct list structure, function returns 1, otherwise returns 0.

For *expr* as an empty string function return value 1.

If list element of *expr* is a list structure too, function count this element as one correct element and does not parse it's internal structure.

Examples:

```
USER>w $lv("string")
0
USER>w $lv($lb())
1
USER>w $lv($lb(1,2,3))
1
USER>w $lv(123)
0
USER>
```

Function $listvalid() check entire value, so some ov possible strings may be used in list operations, but does not be valid lists:

```
USER>s list=$lb(123,456,789)_"string"

USER>w $li(list,2)
456
USER>w $lv(list)
0
```

Function *$listvalid()* and internals of list structures does not included into current MUMPS Standard.

5.25 $NAME

Return string with name of local, global or structured system variable.

Syntax

$NA[ME](glvn[,level])

Definition

glvn	Local, global or structured system variable name.
level	Integer with subscript level.

Function *$name()* return canonical string respresentation of specified variable name with all specified subscripts up to *level* subscripts.

Function *$name()* does not check variable existence and database existence for global variables. Global variables can be specified using naked indicator, and all variables can be specidief using name and subscript indirection syntax. Result of function *$name()* always have canonical representation.

If *level* argument is omitted, function creates name with all specified subscripts.

If value of *level* evaluates as 0, function return variable name only, without subscripts. If value of *level* evaluates less than 0, function generate an error <FUNCTION>.

If value of *level* evaluates greater than really specified subscripts, function return full name with all specified subscripts.

Examples:

```
>s ^A(1,2,3)=1
>w $na(^A(1,2,3))
^A(1,2,3)
>w $na(^A(1,2,3),1)
^A(1)
>w $na(^(2,9,10))
^A(1,2,2,9,10)
TEMP>w $na(^A(1,2,3),2)
^A(1,2)
TEMP>w $na(^A(1,2,3),0)
^A
TEMP>w $na(^A(1,2,3),-1)

<FUNCTION>
```

Note: global variable ^A(2,9,10) here **does not defined**.

5.26 $ORDER

Return next available variable subscript.

Syntax

$O[RDER](name[,direction])

Definition

name	Local, global or structured system variable.
direction	Optional, direction to enumerate next subscript.

Function *$order()* with local and global variable return next available subscript with index sorting, and enumerate from last specified in *name* subscript.

```
TEMP>s a(12)=12,a(45)=45,a("ab")="ab",a("cd")="cd"

TEMP>w $o(a(1))
12
TEMP>w $o(a(100))
```

```
ab
TEMP>w $o(a("bb"))
cd

TEMP>w $o(a("zz"),-1)
cd
TEMP>w $o(a("bb"),-1)
ab
TEMP>w $o(a("100"),-1)
45
TEMP>w $o(a("20"),-1)
12
```

If next subscript does not exist, function return empty string. If last specified in *name* subscript is an empty string, function return last available of first available subscript depending or specified *direction*.

```
TEMP>w $o(a(""),-1)
cd
TEMP>w $o(a(""))
12
```

Optional *direction* can be omitted, or evaluates to 1 or -1. Other values are reserved by standard fo future use and now generates an error <FUNCTION>. Values of *direction* can be one of the following:

1	Return next subscript forward.
-1	Return next subscript backward.
omitted	Return next subscript forward.

If *$order()* function applied to global variable without subscripts, function generates an error <FUNCTION>.

If *$order()* function applied to local variable without subscripts, function return next available local variable name without subscript. This allow enumerate all available local variables by name. Note, *name* need to be correct syntax variable name, but may not exist. For example:

```
TEMP>s a="a",b="b",c="c"

TEMP>s %="%" f  s %=$o(@%) q:%=""  w %,"=",@%,!
```

```
a=a
b=b
c=c

TEMP>s %="zzz" f  s %=$o(@%,-1) q:%=""  w %,"=",@%,!
c=c
b=b
a=a
%=%
```

If function *$order()* applied to global variable, function have a side effect, change naked indicator. If next subscript exists, function changes naked indicator to this existing name, otherwise function changes naked indicator to source specified *name* even this variable does not exist.

```
TEMP>k ^a w $o(^a("")),!,$zr

^a("")
TEMP>s ^a(1)=1 w $o(^a("")),!,$zr
1
^a(1)
TEMP>w $o(^a(1)),!,$zr

^a(1)
```

If argument *name* been specified as global name with database name, naked indicator changes to contain full name with database name too.

```
TEMP>w $o(^|"TEMP"|a(1)),!,$zr

^|"TEMP"|a(1)
```

Function *$order()* with structured system variable depends of this variable meaning.

$order(^$LOCK(name)[,direction]) return available locks, created by any process.

$order(^$ROUTINE(name)[,direction]) return available routine names with source code.

$order(^$DEVICE(name)[,direction]) return device names opened by current process.

$order(^$GLOBAL(name)[,direction]) return global names in current database.

$order(^$JOB(job)[,direction]) return available job's numbers of current MiniM Database Server instance.

5.27 $PIECE

Return substring selected by delimiter.

Syntax

$P[IECE](String,Delimiter[,Start[,End]])

Definition

String	Expression with source string.
Delimiter	Expression with delimiter.
Start	Start piece position.
End	End piece position.

Function select from source *String* substring, using specified *Delimiter* (may be one or more characters). Supposed, source string consist of several pieces, delimited by *Delimiter* and each this piece have position in left-to-right order. Count of this pieces can be evaluated by *$length(String,Delimiter)* function.

Start optional argument defines first piece position need to be selected and *End* optional argument defines last piece position need to select. If *End* argument is specified, function return pieces delimited by *Delimiter*.

If source *String* evaluates as an empty string, function always return an empty string. If *Delimiter* evaluates as an empty string, function always return empty string.

If *Start* argument does not specified, function return first available piece (meaning by default).

If *End* argument is not specified, it supposed equal to *Start* and function return only one piece.

If *Start* evaluates as a position greater than available piece positions in *String*, function return an empty string. If *Start* evaluates as a number less than 1, function suppose it equal 1 and return pieces from first available position.

If *End* evaluates as a number greater than available piece position in *String*, function return only available pieces without padding from specified *Start* position. And, if *End* evaluates less than *Start* argument, function return an empry string.

If *String* expression value starts with *Delimiter*, function suppose that first piece is an empty string. And, if *String* terminates by *Delimiter*, function suppose last piece exist and is an empty string too.

Examples:

```
>w $piece("a^b^c","^")
a
>w $p("a^b^c","^",2)
b
>w $p("a^b^c","^",2,3)
b^c
```

5.28 $QLENGTH

Return subscripts count of variable name.

Syntax

$QL[ENGTH](name)

Definition

name	Expression with local, global or structured system variable name.

Function *$qlength()* return subscripts count of variable name. If variable name have not subscripts, function return value 0.

Examples:

```
TEMP>w $ql("a")
0
TEMP>w $ql("a(1)")
1
TEMP>w $ql("a(1,1)")
2
TEMP>w $ql("a(1,1,1)")
3
```

Global variable name can be specified with and without database name, this does not affect to *$qlength()* retrun.

```
TEMP>w $ql("^a(1,1,1,1)")
4
TEMP>w $ql("^|""abc""|a(1,1,1,1)")
4
```

Database name can be specified inside vertical braces and inside square brackets.

```
TEMP>w $ql("^|""123""|a(1,1,1)")

<FUNCTION>
TEMP>w $ql("^|abc|a(1,1,1)")

<FUNCTION>
```

For structured system variable real variable existence does not checked to exists.

```
TEMP>w $ql("^$lll(1,1,1,1)")
4
TEMP>w $ql("^$abcd(1,1,1,1)")
4
```

For local and global variables real variable existence does not checked, function does not call this variable data or locks and operate only by name. If name contains more than maximum allowed subscripts (63), function generate an error <FUNCTION>.

5.29 $QSUBSCRIPT

Return specified name part, name database name or subscript value.

Syntax

$QS[UBSCRIPT](namevalue,position)

Definition

namevalue	Expression with variable name.
position	Number or name part to return.

Function *$qsubscript()* return part of variable name, name, database name, subscript value from specified in *namevalue* full variable name. Real variable does not checked to exists, locked or contain data. Argument *position* specify real name part to return and can be one of the following:

< -1	Function generate an error <FUNCTION>, this values are reserved by standard for future use.
-1	Function return database name if *namevalue* is a global name. otherwise return an empty string.
0	Function return unsubscripted variable name.
<= n	If *position* from 1 to *n* where *n* is a subscripts count, function return subscript value.
> n	If *position* is greater than subscripts count, function return an empty string.

Examples:

```
TEMP>w $qs("^|""abc""|aaa(1,2,3)",-2)

<FUNCTION>
TEMP>w $qs("^|""abc""|aaa(1,2,3)",-1)
abc
TEMP>w $qs("^|""abc""|aaa(1,2,3)",0)
^aaa
TEMP>w $qs("^|""abc""|aaa(1,2,3)",1)
1
TEMP>w $qs("^|""abc""|aaa(1,2,3)",2)
2
TEMP>w $qs("^|""abc""|aaa(1,2,3)",3)
3
TEMP>w $qs("^|""abc""|aaa(1,2,3)",4)

TEMP>w $qs("^$aaa",0)
^$aaa
```

For global variable names database name can be specified using vertical braces (||) and square brackets ([]).

For structured system variable name does not checked is name exist and implemented or not.

If argument *position* specify more than supported subscripts count (maximum 63), function generate an error <FUNCTION>.

```
TEMP>w $qs("^[abc]aaa(1,2,3)",0)

<FUNCTION>
USER>w $qs("^[""abc""]aaa(1,2,3)",0)
^aaa
```

5.30 $QUERY

Return next available name with subscripts have data.

Syntax

$Q[UERY](name[,direction])

Definition

name	Local, global, or structured system variable name with or without subscripts.
direction	Direction to find out next available name with data, forward or backward.

Function *$query()* return next variable with the same name and have any data defined in index order.

Examples:

```
TEMP>s ^a(1)=1,^a(2,2)=22,^a(3,3,3)=333

TEMP>zw ^a
^a(1)=1
^a(2,2)=22
^a(3,3,3)=333

TEMP>w $q(^a(1))
^a(2,2)
TEMP>w $q(^a(2))
^a(2,2)
```

```
TEMP>w $q(^a(3))
^a(3,3,3)
TEMP>w $q(^a(5),-1)
^a(4)
TEMP>w $q(^a(4),-1)
^a(3,3,3)
TEMP>w $q(^a(3),-1)
^a(2,2)
TEMP>w $q(^a(2),-1)
^a(1)
```

Argument *direction* specify enumerating order - forward or backward. Value of *direction* can be one of the following:

-1	Return next available name backward.
1	Return next available name forward.
omitted	Return next available name forward.
other	Function generate an error <FUNCTION>, values are reserved by stabdard for future use.

If last subscript of *name* is an empty string, function return last of first variable name depending of *direction* value.

```
TEMP>w $q(^a(""))
^a(1)
TEMP>w $q(^a(""),-1)
^a(4)
```

If next variable name does not exist, function *$query()* return an empty string. Exmaples:

```
TEMP>w $q(^a(5))

TEMP>w $q(^a(1),-1)

TEMP>
```

If *name* contains a database name for a global variable, function *$query()* return result with database name too. If database name specified as an empty string, it is equal current database name.

```
TEMP>w $q(^a(1))
^a(2,2)
TEMP>w $q(^|""|a(1))
^|""|a(2,2)
TEMP>w $q(^|"TEMP"|a(1))
^|"TEMP"|a(2,2)
```

Function *$query()* always check variable name contains data and return only existing variable names (function *$data()* for this names return values 1 or 11).

Function *$query()* have a side effect and change naked indicator if *name* is a global variable name. If next name exist, naked indicator chenges to this name, otherwise naked indicator changes to source *name* even if this name does not contain data.

Examples:

```
TEMP>w "next=",$q(^a(1)),!,"naked=",$zr
next=^a(2,2)
naked=^a(2,2)
TEMP>w "next=",$q(^a(5)),!,"naked=",$zr
next=
naked=^a(5)
```

5.31 $RANDOM

Return pseudorandom number in specified interval.

Syntax

$R[ANDOM](Integer)

Definition

Integer	Expression with interval size to return pseudorandom number from 0 to *Integer*.

Function *$random()* return pseudorandom integer number which equal or greater than 0 and less than value of *Integer*. If value of *Integer* evaluates as 1, function always return 0. If value of *Integer* evaluates less then 1, function generate an error <FUNCTION>.

Examples:

Command	**Function result**
`SET X=$RANDOM(25)`	Value of X is from 0 to 24 inclusive.
`SET X=$R(2)`	Value X may be 0 or 1.
`SET X=$R(1)`	Value of X always is 0.
`SET X=$R(-1)`	Error <FUNCTION>.

5.32 $REPLACE

Replaces substring in string and return result.

Syntax

$REPLACE(string,old,new[,start[,count[,case]]])

Definition

string	Expression evaluated as string whithin need to replace substring
old	Substring to replace from
new	Substring to replace to
start	Optional argument, position of symbol to search substring from.
count	Optional argument, count of replaces need to do.
case	Optional argument, specify case sensitivity to search *old*

Function *$replace()* return string based on the *string* with replacing entries *old* to *new* with specified options.

If source *string* evaluates as an empty string, function does nothing and return an empty string.

If value of *old* evaluates as an empty string, function does nothing and return value of source *string*.

If value of *new* evaluates as an empty string, function returns value of source *string* with removing *old* substrings using specified options.

If argument *start* is present, it is start position of symbol to start search *old* in *string*. Any value evaluated as zero or negative number starts search from *string* start. If value of *start* evaluates as an integer greater than *string* length, function does nothing and return source *string*. If argument *start* does not specified, function start search from first *string* symbol.

If argument *count* is specified, this expression evaluaates as an integer and used as a count to search *old* entries. If it is evaluates as -1, this mean

replace all available entries after *start* symbol. If value of *count* evaluates as 0 or any other negative number, function does nothing and return source *string*. If argument *count* is omitted, function replace all entries from *start* symbol.

If argument *case* is specified, it is case-sensitivity option. If argument is omitted or evaluates as zero, function use case sensitive search, otherwise (specified and evaluates as nonzero) function search case insensitive. Note, case sensitivity for current database instance is defined in national collation file, file minim.ini, section Server, key Locale. To compare two symbols case insensitive function shift to the same case both symbols.

Examples:

```
USER>w $replace("qqwweerr","w","AA")
qqAAAAeerr
USER>w $replace("qqwweerr","ww","AA")
qqAAeerr
USER>w $replace("qqwweerr","www","AA")
qqwweerr
USER>w $replace("qqwweerr","w","AA",3)
qqAAAAeerr
USER>w $replace("qqwweerr","w","AA",4)
qqwAAeerr
USER>w $replace("qqwweerr","w","AA",3,1)
qqAAweerr
USER>w $replace("qqwWeerr","w","AA",1,-1,0)
qqAAWeerr
USER>w $replace("qqwWeerr","w","AA",1,-1,1)
qqAAAAeerr
```

5.33 $REVERSE

Return symbols of string in reverse order.

Syntax

$RE[VERSE](String)

Definition

String	Expression to reverse symbols order.

Function *$reverse()* evaluates argument *String* as string and return string with the same symbols in reverse order.

Examples:

Command	**Function result**
`SET X="ABCDEF"`	`$REVERSE(X)="FEDCBA"`
`SET X=""`	`$RE(X)=""`
`SET X=2*64`	`$RE(X)=821`

5.34 $SELECT

Return value evaluated depending of series of conditions.

Syntax

$S[ELECT](Expr:Value[,Expr:Value[,...]])

Definition

Expr	Expression evaluated as boolean condition.
Value	Expression evaluated to return for true condition.

Function *$select()* evaluates arguments in left-to-right order as pairs of *Expr* and *Value* and search first occurence when *Expr* evaluates as nonzero number. Function stop scan arguments and return *Value* of this pair.

Number of pairs can be up to 255.

If function does not find any pait with true *Expr*, function generate an error <SELECT>.

Note: To exclude error condition it is recommended to make at least last case with true condition. For example:

```
$s(a<0:1,a>0:-1,1:0)
```

Examples:

```
>s a=-123
>w $s(a<0:1,a>0:-1,1:0)
-1
>s b=$s(a<0:"one^Rout",a>0:"two^Rout",1:"three^Rout")
>d @b
```

5.35 $STACK

Return information about stack context of current process.

Syntax

$ST[ACK](Level[,Context])

Definition

Level	Stack level number to examine.
Context	Stack context.

Function *$stack()* return information about current process stack context. Stack context number to examine is specified by *Level* argument and can be from 0 (top stack level) to current level, which return system variable *$stack* inclusively.

For 1-argument form *$stack()* function return string with error if was error on this stack level or one of the following codes:

DO	If this stack context has been created by *do* command execution with or without argument.
XECUTE	If this stack context was created by *xecute* command execution.
$$	If this stack context was created by function call with return context.

For 1-rgument form of *$stack()* function are reserved 2 special values:

-1	Returns maximum available stack levels for analisys.
0	Returns eecution context of zero level and process start method

Prcess start method is comma-delimited with execution context and can be the following:

XECUTE,CONSOLE	Process was run in console mode.
XECUTE,TELNET	Process was run for telnet connection.
XECUTE,STD	Process was run with input-output redirection.
XECUTE,JOB	Process was run as background job.

XECUTE,JOBTCP	Process wa srun as background job with accepted TCP/IP socket.
QUIT,MONO	Expression evaluation in the MiniMono context.
XECUTE,MONO	Command line execution in the MiniMono context.
XECUTE,UNKNOWN	Process in command line execution, but start method was not detemined.

For 2-argument form *$stack()* function argument *Context* can be one of the following:

PLACE	Function return routine place executed.
MCODE	Function return code line executed.
ECODE	Function return string with error if was error.

If stack context created by *xecute* command, function return for *Context* = "PLACE" string with one symbol "@".

If executed line has no label, function return for *Context* = "PLACE" string with label + offset + routine name. Label and offset counts from last passed label.

Function *$stack()* can be executed in any context. If argument *Level* is less than -1, function generate an error <FUNCTION>. If argument *Level* is greater than return of $STACK(-1) (maximum available stack frames information), function returns an empty string.

The $STACK() function keeps information about stack frames if process in error state until assignment

```
set $ecode=""
```

and next return real process's stack information.

5.36 $TEXT

Return line of routine source code.

Syntax

$T[EXT]([Label][+Offset][^Routine])
or

$T[EXT](@Expression)

Definition

Label	Label name (subroutine name).
Offset	Line offset relative label.
Routine	Routine name.
Expression	Expression evaluated as string and containing *Label*, *Offset* and *Routine* as need. It is *$text()* function indirection.

Function return string with line of source code of specified routine. If source code is absent, function search line of code inside bytecode. Line of code inserted by compiler if contain double comment (;;).

Line place specified by combination of *Label*, *Offset* and *Routine* and return line with *Offset* relative *Label* inside *Routine*. And value of *Offset* need to be positive integer. First function form requires at least one element of *Label*, *Offset* or *Routine*. Second function form (*Expression*) must contain correct first form.

Function *$text()* return line of code using currently available routine source code. If source code is absent, function in common case return empty string if this line does not contain double comment (;;).

If argument contain *Routine* only, function return first line of code for this routine.

If argument contain *Offset* but does not contain *Label*, function counts line of code from routine's first line.

If *Routine* is specified, but *Offset* is equal 0, function return this routine name.

If *Offset* does not specified, function return line with this *Label*.

If *Routine* does not specified, function suppose it is current routine. For example, *$text(+0)* return currently executed routine.

If function cannot find line of code with specified arguments, function return an empty string.

Routine can be specified using extended syntax, with database name.

Function return string without carriage return symbol.

Examples:

```
=TstRout - source routine text=======
a(num)
 n a,b,c
```

```
 s a=1
 s b=2
 s c=3
 w "current routine offset "num," line is ",$t(+num),!
 w "current routine label offset ",num," line is ",$t(a+num),!
=========

>w $t(^TstRout)
a(num)
>w $t(+2^TstRout)
 n a,b,c
>d a^TstRout(4)
current routine offset 4 line is  s b=2
current routine label offset 4 line is  s c=3
>s param="a+2^TstRout"
>w $t(@param)
s a=1
```

5.37 $TRANSLATE

Return string with symbols substitution.

Syntax

$TR[ANSLATE](String,Old[,New])

Definition

String	Expression evaluated as a string to execute substitution.
Old	Expression evaluated as a string with symbols substitute from.
New	Expression evaluated as a string with symbols substitute to.

Function *$translate()* scan every symbol in *String* in left-to-right order and substitute symbol if it present in *Old* string to symbol from *New* in the same position. If symbol from *Old* have not appropriate symbol in *New* string, this symbol replaced by empty string (removed). If *New* argument does not specified, each symbol of *Old* is removed from *String*. If *New* string have length greater than *Old*, odd symbols are ignored.

Examples:

Function	Function result
`$TR("ABCD","A","X")`	"XBCD", symbol A replaced to X
`$TR("ABCD","AB","X")`	"XCD", Symbol A replaced to X and B to empty string
`$TR("ABCD","AB","ab")`	"abCD"
`$TR("12-34-56","-")`	"123456", removing symbol "-"

5.38 $VIEW

5.38.1 $VIEW("db")

If first argument of *$view()* function evaluates as a "db" string case insensitive, this group of function create, delete, add to data files data blocks. Direct functions can no check configuration file.

$view("db",1,filename,pageblocks)

Create new root data file with *filename* as file name and *pageblocks* blocks number. One block of pages have size 1Mb.

$view("db",2,dbname[,pageblocks])

Add to database *dbname* specified number of page blocks *pageblocks*. If number of *pageblocks* is not specified, function add 1 block of pages. One block of pages have size 1Mb.

If database have growing limit, function does not exceed this limit. If this database does not exist or not mounted, or read only, function generate an error <FUNCTION>.

$view("db",3,filename,pageblocks)

Add to data file *filename* specified number of page blocks *pageblocks*. One block of pages have size 1Mb.

$view("db",4,filename)

Return number of page blocks in specified data file.

$view("db",5,nspace,nblock)

Read and return value of page in this database.

$view("db",6,nspace,nblock,value)

Write value of page *nblock* in this database.

$view("db",7,nspace,nblock)

Lock page *nblock* to read in this database.

$view("db",8,nspace,nblock)

Lock page *nblock* to write in this database.

$view("db",9,nspace,nblock)

Remobe read lock for *nblock* page in this database.

$view("db",10,nspace,nblock)

Remove write lock for *nblock* page in this database.

$view("db",11,nspace,nblock,locktype)

Convert lock for this *nblock* page to specified: 0 - to read, 1 - to write.

$view("db",12,nspace,blocktype)

Allocate new page with *blocktype* type.

$view("db",13,nspace,nblock)

Free *nblock* page from this database.

$view("db",14)

Remove all pages locks in this database.

$view("db",15)

Return list of mounted database with "*" delimiter.

$view("db",15,dbname)

Return existence of database *dbname*: 1 if exist, 0 if not.

$view("db",16,options,filename)

Create backup all mounted databases into file *filename* with backup options *options*.

$view("db",16,options,filename,dblist)

reate backup all databases listed in *dblist* delimited by comma into file *filename* with backup options *options*.

The *options* argument evaluates as a string and need to be a set of special symbol flags:

F or f	Make full backup, cannot be used with D flag
D or d	Make differential backup, cannot be used with F flag
T or t	Truncate journal after backup
R or r	Show full backup report.

If flag T is not present, journal does not truncated.

If flags F and D does not specified, function make full backup.

The *filename* argument evaluates as a string and specify backup file name to backup to. If intermediate directories does not exist, it is creates automatically.

If function succedded ok, it return an empty string. Otherwise function return error code, one of the following:

1	Failed to use bot F and D flags.
2	Specified unsupported beckup flag.
3	Backup file name is much long to use.
4	Backup file name is empty string.
5	Backup file name contains illegal symbols.
6	One of specified to backup database does not mounted.
7	One of specified to backup database is autocreted database. Backup does not support autocreated database.
8	One of specified database does not exist in current configuration.
9	Function failed to find at least one database to backup.
10	Failed to create intermadiate directories.
11	Failed to open file name for writing to backup to.
12	Failed internal file compressor.
13	Failed to write one more page to backup.

If specified journal trancation option and now still work at least one process in transaction state, journal truncation option is ignored, and function $v("db",16) return empty string with following comma and current process count in transaction state.

$view("db",17,filename)

Function return backup file header in the form:

```
Number , Text
```

If *Number* is not zero, it was an error and *Text* contain error text, otherwise *Text* is comma delimited header fields in the following order:

String "MiniM backup file", show it is a MiniM backup file.

String "FULL" of "DIFF", as beckup type - full or differential.

Internal integer transaction sequence number.

Count of database backed up in this file.

Backup date in format dd.mm.yyyy.

$view("db",18,filename)

Function return backed up databases list comma delimited, If return start with a number, it is an error code, comma, and following error text was while access backup file.

$view("db",19,options,filename)

Restore databases from backup file.

Restore options *options* evaluates as a string and need to be a set of special flags:

J or j	Apply journal after restore from backup
S or s	Don't touch journal
F or f	Check this backup file if full backup
D or d	Check this backup file is differential backup
R or r	Report what happens

Backup file name to restore from is specified in *filename* argument.

If database restore succeeded ok, function return an empty string. Otherwise return error code:

1	Failed to use bot alternative restore options.
2	Specified unsupported restore option.
3	Specified illegal file name.
4	Failed to open backup file to read.
5	File is not MiniM backup file.
6	Failed to read from backup file.
7	CRC check failed.
8	File is not full backup file.
9	File is not differential backup file.
10	Still active processes in transaction state.
11	Database listed in backup file, does not exist on current MiniM instance.
12	Database listed in backup file, does not mounted on current MiniM instance.
13	Database listed in backup file is in readonly state.
14	Failed to access journal file.

15	Failed to find last journal file.
16	Failed to find restore point in journal.
17	Failed to read journal file.
18	Failed to write page to database.
19	Failed to write to journal file.
20	Failed to extend one or more database.

On the MiniM Database Server only one process can run database restore function. While this function doing, server suspend execution of other processes and creation of new processes. Server suppose that database restore can be made only by one operator.

To restore database server requires suspend execution all other processes and suspend creation new processes. This executes on the special restore phase. After restore phase complete, suspended processes runs again and server allow create new processes. However MiniM Database Server requires that no any processes still in transaction state during restore.

If flag "R" is specified, function writes out to current device detailed messages what happens.

If flag "J" is specified, function after restoring database blocks search in journal special backup point and apply journaled after this point records to database. Note, existing this journal records after beckup depends of databas configuration.

If does not specified flag "J" and "S", function execute pages restoring and rolls back uncompleted transactions by journal to backup point. It is recommended to restore databases to backup moment.

If flag "S" is specified, functin skip journal handling phase and restore pages only. It is recommended if need to restore from full and several differentioal backup files to save database changes order.

Database restore operator who run database restore functions need to understand beckup and restore functionality and rules to implement server failover. This documents are detailed in the "MiniM advanced guide".

$view("db",20,dbname,nblock)

Check CRC for *nblock* page in *dbname* database. If this database does not exist, function generate error <FUNCTION>. If *nblock* is negative, function generate error <FUNCTION>. If CRC check ok, function return 1, otherwise return 0.

$view("db",21,dbname)

Check CRC of all available pages in this database *dbname*. If this database does not exist, function generate error <FUNCTION>. If at least one page have incorrect CRC, function return 0, otherwise return 1. Function sequentially check all pages are available in this database.

$view("db",22,dbname)

Function return current database size in megabytes and growing limit in megabytes comma delimited. If database have not growing limit, function return only database size and comma. If this database does not exist, function generate error <FUNCTION>.

$view("db",23,readonly)

Function sets *readonly* boolean value to all databases mounted now. If *readonly* evaluates as 0, all databases get write permissions, if nonzero, all databases goes to read only. Function return an empty string.

$view("db",24,dbname)

Function return "ReadOnly" state for specified dbname database.

$view("db",24,dbname,readonly)

Function return current "ReadOnly" state for *dbname* database and apply specified in *readonly* "ReadOnly" state.

5.38.2 $VIEW("dev")

If first argument of *$view()* function evaluates as a "dev" string case insensitive, if is group of functions to support extended device functionality.

$view("dev",1)

Return socket number for current device if it is TCP device, otherwise function generate an error <FUNCTION>.

$view("dev",1,devname)

Return socket number for specified device if it is TCP device, otherwise function generate an error <FUNCTION>.

$view("dev",2)

Return accepted socket number for current device if it is TCP device, otherwise function generate an error <FUNCTION>.

$view("dev",2,devname)

Return accepted socket number for specified device if it is TCP device, otherwise function generate an error <FUNCTION>.

$view("dev",3,devname)

If this device is a TCP or TNT device, function return remote computer name connected to. If device does not connected or is not TCP or TNT device, function return an empty string.

$view("dev",4,devname)

If this device is a TCP or TNT device, function return remote computer IP address connected to. If device does not connected or is not TCP or TNT device, function return an empty string.

$view("dev",5,mask,timeout)

If current device is a COM device, function waits COM port event specified by the mask *mask*. Optional argument *timeout* specify event waint timeout in seconds. If current device is not COM device, function generate an error <FUNCTION>.

Value of *mask* evaluates as an integer and need to be a sum of event flags, flags can be the following:

#0040	A break was detected on input.
#0008	The CTS (clear-to-send) signal changed state.
#0010	The DSR (data-set-ready) signal changed state.
#0080	A line-status error occurred (frame, overrun, parity).
#0100	A ring indicator was detected.
#0020	The RLSD (receive-line-signal-detect) signal changed state.
#0001	A character was received and placed in the input buffer.
#0002	The event character (specified by EVTCHAR option) was received and placed in the input buffer.
#0004	The last character in the output buffer was sent.

If timeout expired function return value 0. If one or more specified events occurs, function return sum of occured events flags.

Masks are single bits in separate positions of binary representation of integer. And common integer value is a sum of all available masks. To select special bit we can use arithmetic operators \ and #. For example, let it be sum of two numbers: #10+#20 (16+32). And, to determine separate bits we can use operators (here use hexadecimal numbers):

```
USER>w (#10+#20\#10#2)
```

```
1
USER>w (#10+#20\#20#2)
1
USER>w (#10+#20\#40#2)
0
USER>w (#10+#20\#8#2)
0
```

Here was determined that inside integer 48 are present masks #10 and #20, but are not presents masks #40 and #8. To distinquish *write #* command and hexadecimal numbers, *write* arguments are placed into parenthesis.

$view("dev",6,mask)

If currende device is a COM device, function sets up or control port signals. If current device is not COM device, function generate an error <FUNCTION>.

Value of *mask* argument evaluates as an integer and need to be an operation code:

6	Clears the DTR (data-terminal-ready) signal.
4	Clears the RTS (request-to-send) signal.
5	Sends the DTR (data-terminal-ready) signal.
3	Sends the RTS (request-to-send) signal.
1	Causes transmission to act as if an XOFF character has been received.
2	Causes transmission to act as if an XON character has been received.
8	Suspends character transmission and places the transmission line in a break state until $view("dev",6,mask) with mask = 9 is called.
9	Restores character transmission and places the transmission line in a nonbreak state.
7	Reset device if possible.

Function return empty string.

$view("dev",7)

If current device is COM device, function return masked state of modem signals (if modem is connected to port). If current device is not COM device, function generate an error <FUNCTION>.

Return value is a sum of masks:

#10	The CTS (clear-to-send) signal is on.
#20	The DSR (data-set-ready) signal is on.
#40	The ring indicator signal is on.
#80	The RLSD (receive-line-signal-detect) signal is on.

$view("dev",8)

If current device is a FILE device, function return current file offset. Otherwise functino generate an error <FUNCTION>.

$view("dev",8,devname)

If specified device is a FILE device, function return current file offset for this device. Otherwise functino generate an error <FUNCTION>.

5.38.3 $VIEW("err")

If first argument of *$view()* function evaluates as a "err" string case insensitive, if is group of functions to return extended error information.

$view("err",1)

Function return place of last error occured in the source code as file name * line number. It is debugging information which can help developers to localise and fix error.

$view("err",2,errnumber)

Function return extended error text by error code. Error codes defined by MUMPS standard and returns by system variable *$ecode* with prefix "M". For example, call to undefined local variable occurs an error "M6":

```
$ecode=",M6,"
```

To get error extended text it is need to pass error code as *errnumber*, for example:

```
w $view("err",2,6)
Undefined local variable.
```

If *errnumber* is not MUMPS standard error code or does not supported by MiniM Database Server in current version, function generate an error <FUNCTION>.

$view("err",3)

Function return code returned by last ZDLL call. This value is returned by exported from external dll function. If function executes normally, return value is 0 by default, otherwise it is error and *$zdll()* function generate an error <FUNCTION>. Programmer can use $view("err",3) function to analize what happens with external dll. If external function violates memory access, $view("err",3) function return value -1. After process start last ZDLL return value is 0 even if *$zdll()* function does not called.

5.38.4 $VIEW("file")

If first argument of *$view()* function evaluates as a "file" string case insensitive, if is group of functions to control files and directories of file system.

$view("file",1,name)

Return information about *name*: 1 - it is existing file, 2 - it is existing directory, 0 - not existing file and not existing directory.

$view("file",2,name)

Function return date and time of file / directory changed in *$horolog* format. If *name* does not corresponds any existing file or directory, function return an empty string. Date and time returned use local time.

$view("file",3,name)

Function return date and time of file / directory created in *$horolog* format. If *name* does not corresponds any existing file or directory, function return an empty string. Date and time returned use local time.

$view("file",4,name)

Function return date and time of file / directory last access in *$horolog* format. If *name* does not corresponds any existing file or directory, function return an empty string. Date and time returned use local time.

$view("file",5,name)

Function return file size in bytes. If *name* does not corresponds existing file, function return an empty string.

$view("file",6,name)

Function delete file. If deletion syccessfull, function return 1, otherwise 0. Function does not support name masks.

$view("file",7,name)

Function delete directory. If deletion syccessfull, function return 1, otherwise 0. Function does not support name masks.

$view("file",8,name,newname)

Function rename file or directory from *name* to *newname*. If operation syccessfull, function return 1, otherwise 0. Function does not support name masks.

$view("file",9,name,newname[,FailIfExist])

Function copy file from *name* into *newname*. Argument *FailIfExist* evaluates as an integer and compares with 0. If Argument *FailIfExist* does not specified, it supposed as 1. If value of *FailIfExist* evaluates as nonzero, function fails if file with *newname* name already exists. If *FailIfExist* evaluates as 0, function ignores *newname* file existence and overwrites.

$view("file",10,name)

Function return string with symbols to indicate file or directory attributes. If function failed to determine attributes, it return -1 value. Attribute symbols are:

A	Archive.
C	Compressed.
D	Directory.
H	Hidden.
O	Offline.
R	Read only.
S	System.

$view("file",11,name)

Function return short file name in 8.3 format. If function fails, return is an empty string.

$view("file",12)

Function return string with available drive letters.

$view("file",13,name)

Return indicator of write type of specified *name* location.

0	Function failed to determine drive type.
1	Specified directory does not exist.
REMOVABLE	Drive has removable media.
FIXED	Drive is fixed.
REMOTE	It is remote or network drive.
CDROM	Drive id CD-ROM or DVD-ROM media drive.
RAMDISK	Drive is in-memory RAM disk.

$view("file",14,name)

Function return total size on drive and free size on drive comma delimited. Sizes are returned in bytes. If function fails to determine size, function return an empty string.

$view("file",15,names)

Function delete files and directories with subdirectories recursively. Argument *names* can contain file mask symbols (*,?). Function support several names in *names* argument delimited by $c(0) symbol. If function complete successfully, return 1 otherwise return 0.

$view("file",16,name)

Function create a directory *name*. If function complete successfully, return 1, otherwise return 0.

$view("file",17,names,dir)

Function copies files and directories from *names* into directory *dir*. Argument *names* can contain file masks (*.?) and several names delimited by $c(0) symbol. Argument *dir* can be only one directory. Function copyes files and directories recursively with subdirectories. If function complete successfully, return 1, otherwise return 0.

$view("file",18,names,to)

Function move or rename files listed in *names* into directory *to*. Argument *names* can contain file masks (*.?) and can contain several names delimited by symbol $c(0). Function rename or move files with subdirectories recursively. If function complete successfully, return value is 1, otherwise 0.

$view("file",19,name,attrs)

Function sets to file *name* attributes *attrs*. If function complete successfully, return 1, otherwise 0. Argument *attrs* specify file attributes case insensitive as special symbols:

A	Archive.
H	Hidden.
R	Read only.
S	System.

$view("file",20,commandline,options,dir)

Function runs child Windows process with specified commandline and with *dir* as current child process directory. Function does not wait while process terminates and return id of child process. If function fails, return code is 0.

commandline	Exe file name with command line parameters
options	Options to run process
dir	Current directory for child process

Argument *commandline* is child process filename with command line parameters. Function use first part of *commandline* up to first space as exe name. Function search file to run in /bin subdirectory of MiniM installation, in current directory of MiniM process, in system Windows directories and in directories listed in PATH environment. For example:

```
w $v("file",20,"minimne.exe","s","")
w $v("file",20,"notepad.exe","s","")
```

If argument *options* is specified, function search in *options* symbol "s" or "S". If this symbol is found, function show child process window, otherwise not and run process with hidden window.

If argument *dir* is not empty string, function use it as current directory for child process.

$view("file",21,commandline,options,dir)

Function runs child Windows process with specified commandline and with *dir* as current child process directory. Function waits while process terminates and return process exit code.

$view("file",22,name)

Function return full long file name for file with *name*. If function fails, or this file does not exist, function return an empty string.

5.38.5 $VIEW("jrnl")

If first argument of *$view()* function evaluates as a "jrnl" string case insensitive, if is group of functions for journaling operations and settings.

$view("jrnl",1)

Function return current journal directory. This directory contains special journal files.

$view("jrnl",2)

Return current journal file name.

$view("jrnl",3)

Function switch current journal file to new next name. Next journaling file name created automatically. Previous used journaling file does not used by journal daemon and journal daemon append records only to last used journal file. All journal files can be used for backup, rollback currently uncomplete trunsactions and to restore database. All journal files need to be saved if this operations need to be executed.

$view("jrnl",4,filename)

Function check file *filename* and return 1 if it is MiniM journal file, otherwise return 0.

$view("jrnl",5)

Function send signal to journal daemon to truncate journal. Function only send signal and return immediately. Function return an empty string.

On journal truncation daemon transform journal files (series of files) to keep only need to complete currently uncomplete transactions. And result size of journal files can be small or big depending server activity. Journal daemon truncate journal until special journal record "no transactions". This special journal marker created periodically, on server start, or database restore and on transaction rollback if detected "no transactions" state.

Journal truncation can be made by database administrator manually or automatically with backup. And journal truncation is recommended with full backup, because database files can be restored with backup and journal tail created after backup. To complete operation journal daemon can use disk space not greater than current total size of journal files.

5.38.6 $VIEW("lock")

If first argument of *$view()* function evaluates as a "lock" string case insensitive, if is group of functions to return locking information.

$view("lock",1)

Return memory size is used by locking objects for *lock* commands.

$view("lock",2)

Return memory size total configured to use by locking objects for *lock* commands.

$view("lock",3)

Return memory size is used by locking objects for cache page locks.

$view("lock",4)

Return memory size total configured to use by locking objects for cache page locks.

5.38.7 $VIEW("log")

If first argument of *$view()* function evaluates as a "log" string case insensitive, if is group of functions to do with MiniM Database Server logging.

$view("log",1)

Function return current log file name.

$view("log",1,newfilename)

Function set up new logging file name. File nam eis limited by 255 symbols. Function return previous log file name.

$view("log",2)

Function return current logging chunk size in bytes to switch log file.

$view("log",2,newchunksize)

Function set up new logging chunk size in bytes. Minimum value is 1024 bytes. Function return previous value of logging chunk size.

$view("log",3,arg1[**,arg2...**]**)**

Function write out to log file values of expressions *arg1, arg2,...* in left-to-right order. Function return an empty string.

5.38.8 $VIEW("perf")

If first argument of *$view()* function evaluates as a "perf" string case insensitive, if is group of functions to access MiniM Database Server performance counters.

$view("perf",1)

Function return count of currently supported by MiniM Database Server performance counters. Other $view("perf") functions accept counter number starts from 1.

$view("perf",2,n)

Function return current value of MiniM performance counter with number *n*. If *n* is unsupported number, function generate an error <FUNCTION>.

$view("perf",3,n)

Function return name of MiniM performance counter. If *n* is unsupported number, function generate an error <FUNCTION>.

$view("perf",4,n)

Function return text description of MiniM performance counter. If *n* is unsupported number, function generate an error <FUNCTION>.

MiniMono difference

MiniM Embedded Edition does not exposes performance counters as a Windows performance counters but counters are still accessible by MUMPS routines using $view("perf") functions.

5.38.9 $VIEW("proc")

If first argument of *$view()* function evaluates as a "proc" string case insensitive, if is group of functions to control current process settings.

$view("proc",1)

Function return current process settings how to use null subscripts. If null subscripts allowed, return 1 otherwise 0. On start process got settings from server configuration file minim.ini. Process can change this settings for semself.

$view("proc",1,NullSubscripts)

Function changes current settings how to use null subscripts. If +NullSubscripts'=0 null subscripts allowed, otherwise denied. Function change setting only for semself. Function return previous setting value.

$view(”proc”,2)

Function return current process journaling state. If value is 1, all global changes made by process are journaled, otherwise not. Journal accept records if process is in journaling state and database is in journaling state.

$view(”proc”,2,JournalFlag)

Function changes current process journaling state and return previous settings state. If JournalFlag is nonzero, journaling for process is enabled, otherwise disabled.

$view(”proc”,3,JobNumber)

Function suspend specified MiniM process (job) execution. If specified process currently still in read or other state with timeout, internal timer still active and after unfreezing process this operation can be stopped by timeout expiration. To resume process execution process must call function *$view(”proc”,4,JobNumber)*. This function cannot be used with current process.

$view(”proc”,4,JobNumber)

Resume process execution previously suspended by function *$view(”proc”,3,JobNumber)*. This function cannot be used with current process.

$view(”proc”,5)

Function return current end-of-file handling mode. Function return 1 if process generate <ENDOFFILE> error on en-of-file error. It is default MiniM setting.

$view(”proc”,5,mode)

Function return current end-of-file handling mode and setup new mode. If value is zero, process does not generate <ENDOFFILE> error and process must check *$zeof* variable state.

$view(”proc”,6)

Function return current *bin* directory of MiniM instance. For example:

```
USER>w $view("proc",6)
W:\MiniM\bin\
```

$view(”proc”,7)

Function return current MiniM process directory, for example:

```
USER>w $v("proc",7)
W:\MiniM\bin
```

Return values may differ for other processes if processes runs from logical disks, created by *subst* Windows command, but all values are the same phisical directories.

In the most cases current process directory is the same directory as *\bin* of current MiniM instance.

$view("proc",8)

Return number of environment variables for current process.

$view("proc",8,n)

Return environment variables pair as "name=value" for specified environment variable number. For example:

```
USER>w $v("proc",8,4)
COMPUTERNAME=AUGUST
USER>w $v("proc",8,5)
ComSpec=E:\WINNT\system32\cmd.exe
```

If argument *n* is less than 0 or greater than available, function return an empty string. Function counts environment variables from 0.

$view("proc",9,name)

Function return current environment variable by name. If this variable does not found, function return an empty string. Environment variable name is used case-insensitive, for example:

```
USER>w $v("proc",9,"comspec")
E:\WINNT\system32\cmd.exe
```

If passed environment variable name terminates with symbol "=", function remove this environment variable for current process.

$view("proc",10)

Function suspend execution of all other processes except semself and daemons. Also function suspend new MiniM process creation.

$view("proc",11)

Function resume execution of all other processes and new MiniM process creation.

$view("proc",12)

Function suspend new MiniM process creation. All available processes still work, but new processes on creation still wait permission to run. If current process terminates or halts, new process creation mode is automatically restored to allow to start.

$view("proc",13)

Function allow to start new MiniM processes. By default this mode is on.

$view("proc",14)

Function evaluates and returns value of current process priority.

$view("proc",14,value)

Function sets up new value of process priority and return prior priority value.

Values of process priorities used by function $view("proc",14), are dependend of the target operating system of MiniM Database Server:

Windows version:

-2	IDLE priority, lowest of all possible.
-1	Priority lower than normal.
0	Normal priority, and priority of the MiniM process by default.
1	Priority above normal.
2	High priority.
3	Real-time priority, highest possible priority.

Linux version:

Values of process priority are integer numbers between and including values of -20 and 20. Value 0 is normal and default MiniM process priority. Values lower than 0 are low priority and -20 is lowest. Values above 0 are high priority and 20 is highest possible.

$view("proc",15)

Function returns limit of process number can be run in current MiniM instance. Total limit counts as a licensed number of processes in minim.lic key file plus 3 engineer processes.

$view("proc",16)

Function returns current number of running jobs in current MiniM instance. While this function counts, no any other processes can be run or stop and this functions counts over freezed ^$JOB state. After function return other processes can be run and stop.

5.38.10 $VIEW("rou")

If first argument of *$view()* function evaluates as a "rou" string case insensitive, if is group of functions to operate by compiler to bytecode.

$view("rou","c",rouname)

Function compile routine's source code into bytecode. Function return 0 if compilation successful, otherwise function return error count was detected. Function write out to current device compilation messages. Result of compilation is a special record in ^rOBJ global with routine's bytecode. If compilation successful, routine is ready to execute and is changes in internal routine cache. Other processes which execute this routine can detect routine bytecode changes and generate an error <EDITED>.

$view("rou","s",line)

Function make syntax parsing *line* as line of commands. If code line is syntax correct, function return an empty string, otherwise function return text with syntax error detected. Current values of *$zerror* and *$ecode* does not changes.

Chapter 6

Z - Functions

6.1 $ZABS

Function return absolute value of argument evaluated as a number.

Syntax

$ZABS(arg)

Definition

arg	Expression to evaluate as a number and return absolute value.

Function *$zabs()* evaluates argument as a number and return it's absolute value (module).

```
TEMP>w $zabs("+7.0e+1 degrees")
70
TEMP>w $zabs("-7.0e-1 is approximate value of 135 degrees")
.7
TEMP>w $zabs(456)
456
TEMP>w $zabs(-123)
123
```

If *arg* does not starts with numeric symbols, function evaluates it as 0 value.

```
TEMP>w $zabs("")
```

```
0
TEMP>w $zabs("-")
0
TEMP>w $zabs("abcd")
0
```

6.2 $ZARCCOS

Function return arc cosine in radians.

Syntax

$ZARCCOS(Number)

Definition

Number	Expression evaluated as a number.

Argument *Number* must be between -1 and 1; other values generate an error <ILLEGAL VALUE>.

Examples:

```
TEMP>w $zarccos(-2)

<ILLEGAL VALUE>
TEMP>w $zarccos(0.8)
.643501108793284
TEMP>w $zarccos(0.2)
1.36943840600457
TEMP>w $zarccos(0)
1.5707963267949
```

6.3 $ZARCSIN

Function return arcsine of argument in radians.

Syntax

$ZARCSIN(Number)

Definition

Number	Expression evaluated as a number.

Argument *Number* must be between -1 and 1; other values generate an error <ILLEGAL VALUE>.

Example:

```
TEMP>w $zarcsin(0.7)
.775397496610753
```

6.4 $ZARCTAN

Function return arctangent of argument in radians.

Syntax

$ZARCTAN(Number)

Definition

Number	Expression evaluated as a number.

Function return arctangent of argument in radians. Return values are from -1.57079 ($-\pi/2$) to 1.57079 ($+\pi/2$).

Example:

```
TEMP>w $zarctan(1)
.785398163397448
```

6.5 $ZBITAND

Function return bitwise logical operation over two zbitstrings.

Syntax

$ZBITAND(bitstr1,bitstr2)

Definition

bitstr1	First argument, zbitstring.
bitstr2	Second argument, zbitstring.

Function *$zbitand()* use arguments as zbitstrings, evaluates bitwise AND operation and return result as zbitstring too. If one argument have smaller bit length, function create result using smaller zbitstring and counts absent

bits as 0.

If one of argument does not containt valid zbitstring, function *$zbitand()* generate an error <FUNCTION>.

For example, if bitstr1=[1010], bitstr2=[10000000], function return new zbitstring [1000].

6.6 $ZBITCAT

Function return zbitstring as concatenation of two zbitstrings.

Syntax

$ZBITCAT(bitstr1,bitstr2)

Definition

bitstr1, bitstr2	Expression with zbitstrings values.

Function *$zbitcat()* makes new zbitstring with bits concatenated from argument's zbitstrings *bitstr1* with following *bitstr2*. Result length in bits is equal sub of argument's length in bits.

Examples:

```
USER>s a=$zbitstr(9,1)

USER>s b=$zbitstr(9,0)

USER>s str=$zbitcat(a,b)

USER>f i=1:1:$zbitlen(str) w $zbitget(str,i)
111111111000000000
```

6.7 $ZBITCOUNT

Function return number of true bits available in zbitstring.

Syntax

$ZBITCOUNT(bitstr)

Definition

bitstr	Argument evaluated as a string and containing zbitstring.

Function *$zbitcount()* return number of true bits are available in specified zbitstring. If argument *bitstr* is not valid zbitstring, function generate an error <FUNCTION>. For example, for bitstr = [1100011] function return 4:

```
USER>s str=$zbitstr(7,1)

USER>s str=$zbitset(str,3,0)

USER>s str=$zbitset(str,4,0)

USER>s str=$zbitset(str,5,0)

USER>w $zbitcount(str)
4
```

6.8 $ZBITEXTRACT

Function return zbitstring as a part of source zbitstring.

Syntax

$ZBITEXTRACT(bitstr,from,count)

Definition

bitstr	Source zbitstring to extract bit from.
from	Bit number to extract from. Bitc counted from 1.
count	Number of bits to return in result zbitstring.

Function *$zbitextract()* return zbitstring as a part of source zbitstring from specified in *from* bit and *count* bits length. If source *bitstr* is not valid zbitstring or interval from *from* with length *count* is not a part of source *bitstr*, function generate an error <FUNCTION>.

Examples:

```
USER>s a=$zbitstr(3,1),b=$zbitstr(3,0),str=$zbitcat(a,b)
```

```
USER>f i=1:1:$zbitlen(str) w $zbitget(str,i)
111000
USER>s str=$zbitextract(str,2,3)

USER>f i=1:1:$zbitlen(str) w $zbitget(str,i)
110
```

6.9 $ZBITFIND

Function search and return next position of specified bit.

Syntax

$ZBITFIND(bitstr,bit[,position])

Definition

bitstr	Expression evaluated as a string and used as a zbitstring.
bit	Expression evaluated as 0 or 1 and specified what bit to be search.
position	Optional, specify position to start search from.

Function *$zbitfind()* search specified bit *bit* in source *bitstr* from specified position *position*. If function find in next position this bit, function return a position number after found bit.

If function cannot find specified bit, function return 0.

If argument *position* does not specified, function start search from *bitstr* first bit.

If specified *position* is less then 1 or is greater than maximum allowed ($262128 = 32766 * 8$), function generate an error <FUNCTION>.

Example:

```
TEMP>s str=$zbitstr(5,1)

TEMP>s i=1 f  s i=$zbitfind(str,1,i) q:'i  w i,!
2
3
4
5
6
```

6.10 $ZBITGET

Function return bit value at specified position.

Syntax

$ZBITSTR(bitstr,position)

Definition

bitstr	Expression evaluated as a string to return bit value from.
position	Position in zbitstring to return bit value.

Function *$zbitget()* return bit value (0 or 1) from source *bitstr* as zbitstring at specified position *position*.

Bit positions counts from 1.

If specified *position* is less then 1 or is greater than maximum allowed ($262128 = 32766 * 8$), function generate an error <FUNCTION>.

Example:

```
TEMP>s str=$zbitstr(5,1)

TEMP>f i=1:1:$zbitlen(str) w $zbitget(str,i)
11111
```

6.11 $ZBITLEN

Function return number of bits available in zbitstring.

Syntax

$ZBITLEN(bitstr)

Definition

bitstr	Expression evaluated as a string and used as zdinstring to get number of bits.

Function *$zbitlen()* return number of bits total available in specified zbitstring *bitstr*. Function return number of bits including 0 and 1 bits. If value of *bitstr* is not valid zbitstring, function generate an error <FUNCTION>.

Examples:

```
TEMP>w $zbitlen($zbitstr(9,1))
9
TEMP>w $zbitlen($zbitstr(15,0))
15
```

6.12 $ZBITNOT

Function return zbitstring with inverted bits values.

Syntax

$ZBITNOT(bitstr)

Definition

bitstr	Expression evaluated as a string with zbitstring to invert.

Function *$zbitnot()* creates zbitstring from source *bitstr* and invert all bits available. If source *bitstr* is not valid zbitstring, function generate an error <FUNCTION>.

Examples:

```
TEMP>s str=$zbitstr(25,1)

TEMP>f i=1:1:$zbitlen(str) w $zbitget(str,i)
1111111111111111111111111
TEMP>s str=$zbitnot(str)

TEMP>f i=1:1:$zbitlen(str) w $zbitget(str,i)
0000000000000000000000000
```

6.13 $ZBITOR

Function make bitwise logical OR with two zbitstrings and return result.

Syntax

$ZBITOR(bitstr1,bitstr2)

Definition

bitstr1	First argument, zbitstring.
bitstr2	Second argument, zbitstring.

Function *$zbitor()* use arguments as zbitstrings, evaluates bitwise OR operation and return result as zbitstring too. If one argument have longer bit length, function create result using longer zbitstring and counts absent bits as 0.

If one of argument does not containt valid zbitstring, function *$zbitor()* generate an error <FUNCTION>.

For example, if bitstr1=[1010], bitstr2=[10000000] function *$zbitor()* return result [10100000].

6.14 $ZBITROT

Function return zbitstring with cyclically shifted (rotated) bits.

Syntax

$ZBITROT(bitstr,direction)

Definition

bitstr	Expression evaluated as a string and used as a source zbitstring.
direction	Expression evaluated as a bits shift (rotate) direction.

Function *$zbitrot()* rotate bits of source zbitstring left or right and return result. It is cyclical bits shifting by one. If value of *direction* evaluates as 1, function rotate right, if evaluates as -1, rotate left, otherwise function generate an error <FUNCTION>.

Examples:

Rotate right:

```
USER>s a=$zbitstr(3,1),b=$zbitstr(3,0),str=$zbitcat(a,b)

USER>f i=1:1:$zbitlen(str) w $zbitget(str,i)
111000
USER>s str=$zbitrot(str,1)

USER>f i=1:1:$zbitlen(str) w $zbitget(str,i)
011100
```

Rotate left:

```
USER>s a=$zbitstr(3,1),b=$zbitstr(3,0),str=$zbitcat(a,b)

USER>f i=1:1:$zbitlen(str) w $zbitget(str,i)
111000
USER>s str=$zbitrot(str,-1)

USER>f i=1:1:$zbitlen(str) w $zbitget(str,i)
110001
```

6.15 $ZBITSET

Function return zbitstring with one bit changed.

Syntax

$ZBITOR(bitstr,position,bit)

Definition

bitstr	Expression evaluated as a string and used as a source zbitstring.
position	Expression evaluated as a number with bit position.
bit	New value of bit.

Function *$zbitset()* use argument *bitstr* as a source zbitstring and changes one bit in position *position* to value of *bit*. If source *bitstr* is not valid zbitstring, function generate an error <FUNCTION>. Return value is source zbitstring with one bit changed.

Argument *position* evaluates as an integer, If this value is less then 1 or greater then maximum allowed ($262128 = 32766 * 8$), function generate an error <FUNCTION>. If *position* show outside of available in zbitstring position, function does not expand *bitstr* and generate an error <FUNCTION>.

Value of *bit* evaluates as an integer and compares with 0, if it is 0, function sets bit to 0, otherwise sets bit to 1.

Examples:

```
TEMP>s str=$zbitstr(15,1)

TEMP>f i=1:1:$zbitlen(str) w $zbitget(str,i)
111111111111111
TEMP>s str=$zbitset(str,12,0)
```

```
TEMP>f i=1:1:$zbitlen(str) w $zbitget(str,i)
111111111110111
```

6.16 $ZBITSTR

Function return zbitstring with specified bits count.

Syntax

$ZBITSTR(Count,Bit)

Definition

Count	Count of bits in result zbitstring.
Bit	Value of bits to fit result.

Function *$zbitstr()* creates and return new zbitstring value. Count of bits in result is specified by *Count* and bits value to fit result is specified by *Bit*.

Value of *Count* must be greater than 0 and less than 262128 ($32766 * 8$).

Value of *Bit* evaluates as an integer and compares with 0, if it is 0, result zbitstring fits with 0 bits, otherwise fits with 1 bit.

Examples:

```
TEMP>ZZDUMP $ZBITSTR(-12,2)

<FUNCTION>
TEMP>ZZDUMP $ZBITSTR(500000,2)

<FUNCTION>
```

$zbitstr(15,1) return zbitstring with 15 bits of 1.

$zbitstr(32,0) return zbitstring with 32 bits of 0.

6.17 $ZBITXOR

Function make bitwise logical exclusive OR (XOR) with two zbitstrings and return result.

Syntax

$ZBITXOR(bitstr1,bitstr2)

Definition

bitstr1	First argument, zbitstring.
bitstr2	Second argument, zbitstring.

Function *$zbitxor()* use arguments as zbitstrings, evaluates bitwise XOR operation and return result as zbitstring too. If one argument have smaller bit length, function create result using smaller zbitstring and does not fit absent bits.

If one of argument does not containt valid zbitstring, function *$zbitxor()* generate an error <FUNCTION>.

For example, if bitstr1=[1010], bitstr2=[10000000], function *$zbitxor()* return result [0010].

6.18 $ZBOOLEAN

Function create bitwise operation over arguments and return result.

Syntax

$ZBOOLEAN(arg1,arg2,bitoper)

$ZB(arg1,arg2,bitoper)

Definition

arg1	First argument, string or an integer.
arg2	Second argument, string or an integer.
bitoper	Logical operation code.

Function *$zboolean()* make a bitwise operation over two arguments *arg1* and *arg2*, where operation code is specified by *bitoper* argument. Function return result of bitwise operation.

Function use arguments *arg1* and *arg2* both as integers or both as a strings. If one of argument evaluates as a fractional number, function generate an error <FUNCTION>. If both arguments are integers, result is integer, otherwise both arguments are used as a strings and function make bitwise operation over bytes of strings.

If arguments *arg1* and *arg2* are used as a strings, result is based on *arg1*

length, if operation does not specify other. If length of *arg2* is greater than length of *arg1*, function use only bytes to *arg1* length. If length of *arg2* is less than *arg1*, function apply bytes of argument *arg2* cyclically.

Value of *bitoper* evaluates as an integer and used as value from 0 to 15 inclusively. If value of *bitoper* is less than 0, function use 0, if value is greater than 15, function use only 4 low order bits of *bitoper* and gets value from 0 to 15.

Values of *bitoper* corresponds to bitwise logical operators:

```
 0   0
 1   arg1 & arg2
 2   arg1 & ~ arg2
 3   arg1
 4   ~ arg1 & arg2
 5   arg2
 6   arg1 ^ arg2
 7   arg1 ! arg2
 8   ~ ( arg1 ! arg2 )
 9   ~ ( arg1 ^ arg2 )
10   ~ arg2
11   arg1 ! ~ arg2
12   ~ arg1
13   ~ arg1 ! arg2
14   ~ ( arg1 & arg2 )
15   -1
```

Here are defined operators:

```
&   AND
!   OR
~   NOT
^   exclusive OR
```

If programmer want to guarantee argument using as an integers, it can be done using the "+" (plus) operator, and using argument as a string can be done using concatenation with an empty string.

Examples:

```
USER>w $zb(1,2,3)
1
USER>w $zb(1,2,6)
3
USER>w $zb(1,2,7)
3
USER>w $zb(1,2,12)
-2
USER>w $zb("aa","bb",7)
cc
USER>s xorkey="this is key"

USER>s str1="this is string with data"

USER>s str2=$zb(str1,xorkey,6)

USER>s str3=$zb(str2,xorkey,6)

USER>w str1,!,str3,!
this is string with data
this is string with data
USER>w str2
        ??
          ???SW HK??
```

6.19 $ZCOS

Function return cosine of angle in radians.

Syntax

$ZCOS(Number)

Definition

Number	Exmpression evaluated as a number.

Cosine evaluates for numbers between $-\pi$ and π (may be applied system variable *$ZPI*), all other numbers reduced to a canonical interval.

Example:

```
>w $zcos(-3*$zpi/4)
```

```
-.7071067811865475242
```

Math function operate with numbers with some possible miscalculation.

```
TEMP>w $zcos($zpi/2)
6.12303176911189E-17
```

6.20 $ZCOT

Function return cotangent of angle in radians.

Syntax

$ZCOT(Number)

Definition

Number	Exmpression evaluated as a number.

Function evaluates cotangent for angles between $-\pi$ and π (may be applied system variable *$ZPI*), all other numbers reduced to a canonical interval.

Function *$zcot()* operate with numbers with some possible miscalculation and some arguments can cause errors and some can not.

```
TEMP>w $zcot(0)

<DIVIDE>
TEMP>w $zcot($zpi/4)
1
TEMP>w $zcot($zpi)
-8.16588936419E+15
TEMP>w $zcot($zpi/2)
6.12303176911E-17
```

6.21 $ZCRC

$ZCRC

Function calculate and return checksum.

Syntax

$ZCRC(String,Algorithm)
$ZCRC(String,Algorithm,StartValue)

Definition

String	Expression evaluated as a string to check sum.
Algorithm	Algorithm specification to calculate checksum.
StartValue	Optional, initial value to calculate checksum.

Checksum algorithms

Argument *Algorithm* can be one of the following string:

SUM	8-bit unsigned bytes sum.
XOR	8-bit XOR of the bytes.
16	16-bit CRC-16.
32	32-bit CRC-32.
CCITT	16-bit CRC-CCITT.
MD5	Hash algorithm MD5.
other	Function generate an error <FUNCTION>.

Argument *Algorithm* is used case insensitive.

To calculate checksum of long data with several segments over 32K length total need to be used optional argument *StartValue* to pass checksum for next data segment.

For "MD5" algorithm argument *StartValue* is not supported and function generate an error <FUNCTION>.

Examples:

```
TEMP>w $zcrc("abcdef","XOR")
7
TEMP>w $zcrc("abcdef","XOR",3)
4
TEMP>w $zcrc("abcdef","16")
22533
TEMP>w $zcrc("abcdef","32")
1267612143
TEMP>w $zcrc("abcdef","SUM")
597
TEMP>w $zcrc("abcdef","CCITT")
62329
```

```
TEMP>w $zcrc("abcdef","www")

<FUNCTION>

USER>w $zcrc("","md5")
D41D8CD98F00B204E9800998ECF8427E
USER>w $zcrc("","md5",123)

<FUNCTION>
```

6.22 $ZCSC

Function return cosecant of angle in radians.

Suntax

$ZCSC(Number)

Definition

Number	Exmpression evaluated as a number.

Function evaluates cosecant for angles between $-\pi$ and π (may be applied system variable *$ZPI*), all other numbers reduced to a canonical interval.

Function *$zcsc()* operate with numbers with some possible miscalculation and some arguments can cause errors and some can not.

Example:

```
TEMP>w $zcsc($zpi/4)
1.41421356237309
```

6.23 $ZCONVERT

$ZCONVERT
$ZCVT

Function convert data into or from one of supported encoding format.

Syntax

$ZCONVERT(String,Direction)
$ZCONVERT(String,Direction,Format)

Definition

String	Source value to convert.
Direction	Convert option.

Value of *Direction* can be one of the following:

`"l", "L"`	Convert *String* to lower case.
`"u", "U"`	Convert *String* to upper case.
`"o", "O"`	Convert *String* to output format.
`"i", "I"`	Convert *String* from input format.

Conversion formats

`"URL"`	Output is URL-encoding.
`"XML"`	Replace XML entities.
`"IHEX"`	Output format is integer in HEX encoding.
`"SHEX"`	Output format is string in HEX encoding.
`"BASE64"`	Output is BASE64 encoding (RFC 989)
`"S1"`	Binary signed 1 byte
`"S2"`	Binary signed 2 bytes
`"S4"`	Binary signed 4 bytes
`"S8"`	Binary signed 8 bytes
`"U1"`	Binary unsigned 1 byte
`"U2"`	Binary unsigned 2 bytes
`"U4"`	Binary unsigned 4 bytes
`"F4"`	Binary floating point 4 bytes (single precision)
`"F8"`	Binary floating point 8 bytes (double precision)

Format name is used case insensitive. Function use for uppercase and lowercase conversion current locale used by MiniM instance. If function detect unsupported formats or invalid encoding, function generate an error. Binary conversion use little-endian encoding (Intel CPU byte order).

Examples:

```
USER>w $zcvt("","u")

USER>w $zcvt("","l")

USER>w $zcvt("","o","url")
%D2%E5%EA%F1%F2
```

```
USER>w $zcvt("22 < 33","o","xml")
22 &lt; 33
USER>w $zcvt("22 < 33","o","shex")
3232203C203333
USER>w $zcvt("123456","o","ihex")
1E240
USER>s base64=$zcvt("Hello, world!","o","base64")

USER>s src=$zcvt(base64,"i","base64")

USER>w
base64="SGVsbG8sIHdvcmxkIQ=="
src="Hello, world!"
USER>zzdump $zcvt("127","o","s1")
0000: 7F
USER>w $zcvt($zcvt("250","o","u1"),"i","s1")
-6
USER>zzdump $zcvt(123.456,"o","f8")
0000: 77 BE 9F 1A 2F DD 5E 40
```

6.24 $ZDATE

Function convert date from *$horolog* format to specified date representation.

Syntax

$ZD[ATE](Value[,Format[,FormatString]])

Definition

Value	Expression evaluated as an integer and used as first part of $horolog date format.
Format	Expression evaluated as an integer and used as format code. Optional, by default 1.
FormatString	Expression evaluated as a string and used as detailed format specification for *Format = 15*

Argument *Value* may be from -672045 (Jan 1, 1) from 2980013 (Dec 31, 9999), and if *Value* evaluates as an integer outside supported interval, function generate an error <ILLEGAL VALUE>.

Argument *Format* specify format of date representation and can be one

of the following:

0	DD Mmm YYYY (08 Mar 2006)
1	MM/DD/YYYY (03/08/2006)
2	DD Mmm YYYY (08 Mar 2006)
3	YYYY-MM-DD (2006-03-08)
4	DD/MM/YYYY (08/03/2006)
5	Mmm D, YYYY (Mar 8, 2006)
6	Mmm D YYYY (Mar 8 2006)
7	Mmm DD YYYY (Mar 08 2006)
8	YYYYMMDD (20060308)
9	Mmmmm D, YYYY (March 8, 2006)
10	W (3)
11	Www (Wed)
12	Wwwww (Wednesday)
13	D Mmmmm YYYY (8 March 2006)
14	DD.MM.YYYY (08.03.2006)

Table of symbols:

D	Day of month (1-31) without leading zero if less than 10.
DD	Day of month (1-31) with leading zero if less than 10.
MM	Double-digited number of month (01-12).
YYYY	Four figures number of year.
Mmm	Short month name.
Mmmmm	Long month name.
W	Week day (0-6, 0-Monday, 1-Tuesday, ...).
Www	Short week day.
Wwwww	Full week day.

If *Format* is equal 15, function use *FormatString* argument to format day. Argument *FormatString* have to specify format symbols starting with %. If after this format specifier follows one of format symbol, function format string as specified. If after % symbol follows % symbol again, function formats only one % symbol. If after % symbol follows unsupported format symbols, function outputs two symbols 00. Table of supported format symbols:

a	Short week day
A	Full week day

b	Short month name
B	Full month name
d	Day of month
H	Hour (24)
I	Hour (12)
j	Day of year
m	Number of month
M	Minute
p	Symbol AM/PM
S	Seconds
U	Year week, when first week starts from sunday
w	Week day.
W	Year week, when first week starts from monday.
y	Year number without century
Y	Year number with century
C	Century
#d	Day of month without leading zeroes.
%	Symbol %

Examples:

```
USER>w $zd($h,15,"Now is %#d day of %m month of %Y year.")
Now is 21 day of 12 month of 2008 year.
USER>w $zd($h,15,"%#d.%m.%Y %H:%S")
21.12.2008 17:31
```

If *Format* argument have unsupported value, function generate an error <FUNCTION>.

Week day names and month names server accept from MiniM service account. To change this values need to start MiniM service under appropriate account and after restart MiniM process will display week day names and month names on new language. Other symbols and format options does not depend of MiniM service account.

6.25 $ZDATEH

Function convert date from formatter string representation into standard *$horolog* format.

Syntax

$ZDATEH(Value[,Format])
$ZDH(Value[,Format])

Definition

Value	String to convert from.
Format	Expression evaluated as an integer and used as format number. Default value is 1.

Argument *Format* specify format of date representation and can be one of the following:

0	DD Mmm YYYY (08 Mar 2006)
1	MM/DD/YYYY (03/08/2006)
2	DD Mmm YYYY (08 Mar 2006)
3	YYYY-MM-DD (2006-03-08)
4	DD/MM/YYYY (08/03/2006)
5	Mmm D, YYYY (Mar 8, 2006)
6	Mmm D YYYY (Mar 8 2006)
7	Mmm DD YYYY (Mar 08 2006)
8	YYYYMMDD (20060308)
9	Mmmmm D, YYYY (March 8, 2006)
13	D Mmmmm YYYY (8 March 2006)
14	DD.MM.YYYY (08.03.2006)

Table of symbols:

D	Day of month (1-31) without leading zero if less than 10.
DD	Day of month (1-31) with leading zero if less than 10.
MM	Double-digited number of month (01-12).
YYYY	Four figures number of year.
Mmm	Short month name.
Mmmmm	Long month name.

If *Format* argument have unsupported value, function generate an error <FUNCTION>. Format codes are the same as format code for *$zdate()* function with the exception of week days and week day names.

Minimum supported day is Jan 1, 1 and maxumum supported day is Dec 31, 9999. If date is outside of this interval, function generate an error

<ILLEGAL VALUE>.

Month names server accept from MiniM service account. To change this values need to start MiniM service under appropriate account and after restart MiniM process will accept month names on new language. Other symbols and format options does not depend of MiniM service account.

6.26 $ZDLL

$ZDLL

Function load, unload and call functions in external dynamic libraries (ZDLL - modules).

Syntax

$ZDLL("load",filename)

$ZDLL("call",filename,funcname[,param,...])

$ZDLL("unload")

$ZDLL("unload",filename)

Definition

filename	Expression with file name of ZDLL.
funcname	Expression with function name in ZDLL.
param	Function parameters.

Function *$zdll()* load, unload and call exported external functions of dynamic libraries with ZDLL interface. Function can pass zero, one or more arguments to external function and return value from ZDLL. All actions with ZDLL module doing with single function *$zdll()* depending of first argument value. This argument used case insensitive and can be one of: "load" - function load and initialize ZDLL module, "unload" - free specified or all ZDLL module, and "call" - call function in ZDLL module, pass parameters and accept return value and error code.

LOAD

To load ZDLL module it is required second argument with dll file name to load. Function search this module inside already loaded and if module is not loaded, function load and initialize it. Function search file in current /bin subdirectory, Windows system directories and in directories listed in PATH envarinment variable. File can have any extension, it's not of necessity to

have dll extension.

If file is found, ZDLL module loads into memory and initializes. Function *$zdll()* import from ZDLL function with name "ZDLL" and call to get all exported by module function list. After this ZDLL still loaded up to direct call to "unload" action or until process terminates.

It is recommended to use dll file names in Windows conventions to access files under all MiniM processes accounts. Most recommended directory is a /bin subdirectory of MiniM instance or special subdirectory for this ZDLL module. Administrator must be sure the MiniM can access this dll files under used Windows accounts.

If "load" action completes successfully, *$zdll()* function return an empty string. If any error occured, function generate an error <FUNCTION>.

UNLOAD

This action may be used in 1-argument and 2-argument forms. In 1-argument form MiniM process unloads all loaded ZDLL modules. In 2-argument form process unloads only specified ZDLL module if this module already was loaded. Second argument must contain dll file name.

CALL

To handle "call" action *$zdll()* function search inside already loaded ZDLL modules this module. If module is already loaded, function call exported function from ZDLL module and module still loaded, and if this module is not loaded, function load, call exported function and unload ZDLL module.

To pass parameters and accept return value *$zdll()* function prepare internal structures in special declared interface. Function *$zdll()* accept variable-length arguments and all arguments after function name will be passed to ZDLL module as values of expressions are evaluated in left-to-right order. After function coll exported function, return value returns into MUMPS code as a function return value.

All arguments to exported ZDLL module functions can be passed by values only.

If exported function return non-success result indicator, *$zdll()* function generate an error <FUNCTION> and last return is accessible using *$view("err",3)* function.

ZDLL exported function get with passed arguments special set of functions to call current MiniM process context to execute commands, evaluate

expression or convert data. Current MiniM process context can be changed by this ZDLL callback functions, and ZDLL module can read or create local or global variables or change values of system variables.

Special ZDLL module description for low-level programmers contains in MiniM Advanced Guide, minimadv.pdf. Samples how to use ZDLL functions are present in MiniM installation subdirectory /zdll.

6.27 $ZEXP

Returns the natural logarithm raised to the specified power.

Syntax

$ZEXP(Number)

Definition

Number	Expression evaluated as a number.

Function *$zexp()* evaluates and returns the natural logarithm raised to the specified power *Number*, or evaluates an exponent of *Number*.

Example:

```
TEMP>w $zexp(1)
2.71828182845905
```

6.28 $ZLASCII

Function converts a four-byte substring of string to a number.

Syntax

$ZLA[SCII](String[,Position])

Definition

String	Expression evaluated as a string to get four-byte substring from.
Position	Expression evaluated as an integer to get four-byte substring from. Optional.

Function *$zlascii()* return integer calculated from four-byte substring from source *String*. Decimal value of a result is converted from ASCII codes of bytes of specified substring. If argument *Position* does not specified, function gets first four-byte substring. If source *String* is an empty string, function return value -1. If from specified *Position* source *String* does not contain four bytes, function return value -1.

Function *$zlascii()* can be represented using standard functions and operators by formula:

$zla(str,pos) = $a(str,pos) + ($a(str,pos+1) * 256) + ($a(str,pos+2) * 256 * 256) + ($a(str,pos+3) * 256 * 256 * 256)

Examples:

```
TEMP>w $zla("123456")
875770417
TEMP>w $zla("123456",3)
909456435
TEMP>w $a(3)+($a(4)*256)+($a(5)*65536)+($a(6)*16777216)
909456435
TEMP>w $zla("123456",5)
-1
TEMP>w $zla("123456",-2)
-1
TEMP>w $zla("")
-1
```

6.29 $ZLCHAR

Function converts a number to a four-byte string.

$ZLC[HAR](Integer)

Definition

Integer	Expression evaluated as an integer to convert from.

Function *$zlchar()* return four-byte string converted from *Integer* using ASCII codes. If argument is less than 0 or is greater than 256∗256∗256∗256 = 4294967296, function return an empty string.

Function *$zlchar()* can be represented using standard functions and operators by formula:

```
$zlc(n) = $c(n#256,n\256#256,n\(256**2)#256,n\(256**3))
```

Examples:

```
TEMP>w $zla("abcd")
1684234849
TEMP>w $zlc($zla("abcd"))
abcd
TEMP>w $zlc(-3)

TEMP>
```

6.30 $ZLCASE

Function convert all symbols to lower case.

Syntax

$ZLCASE(String)

Definition

String	Expression evaluated as a string to convert from.

Function *$zlcase()* return string based on argument with converting all symbols to lower case. Conversion is made using current locale file. See configuration file minim.ini, section Server, key Locale.

Examples:

```
TEMP>w $zlcase("MiniM")
minim
```

6.31 $ZLN

Returns the natural logarithm of the specified number.

Syntax

$ZLN(Number)

Definition

Number	Expression evaluates as a number.

Function *$zln()* returns the natural logarithm of the specified *Number*.

Function with argument 0 or less than 0 generate an error <ILLEGAL VALUE>.

Examples:

```
TEMP>w $zln(0)

<ILLEGAL VALUE>
TEMP>w $zln(-4)

<ILLEGAL VALUE>
TEMP>w $zln(4)
1.38629436111989
TEMP>w $zln("asd")

<ILLEGAL VALUE>
TEMP>w $zln(1)
0
TEMP>w $zln($zexp($zpi))
3.14159265358979
```

6.32 $ZLOG

Returns the base 10 logarithm value of the specified numeric xpression.

Syntax

$ZLOG(Number)

Definition

Number	Expression evaluates as a number.

Function *$zlog()* returns the base 10 logarithm of the specified *Number*.

Function with argument 0 or less than 0 generate an error <ILLEGAL VALUE>.

Examples:

```
TEMP>w $zlog(0)

<ILLEGAL VALUE>
TEMP>w $zlog(-4)

<ILLEGAL VALUE>
TEMP>w $zlog(1)
0
TEMP>w $zlog(10)
1
```

6.33 $ZLOWER

Function convert all symbols to lower case.

Syntax

$ZLOWER(String)

Definition

String	Expression evaluated as a string to convert from.

Function *$zlower()* return string based on argument with converting all symbols to lower case. Conversion is made using current locale file. See configuration file minim.ini, section Server, key Locale.

Examples:

```
TEMP>w $zlower("MiniM")
minim
```

6.34 $ZPCREMATCH

Function verify string match specified regular expression.

Syntax

$ZPCREMATCH(str,regexp[,options])

$ZPCREM(str,regexp[,options])

Definition

str	Expression evaluated as a string to verify.
regexp	Regular expression.
options	Matching options as symbol flags, optional.

Function *$zpcrematch()* verify match *str* argument to specified regular expression *regexp* or not. All arguments can be specified as evaluatable expressions. Argument *options* specify special matching options. If *options* omitted, function suppose it is equal empty string. Function returns value 1 if value of *str* fully match to specified regular expression *regexp*, otherwise function returns 0. If function detect than regular expression has invalid regular expression syntax, function generate an error <PCRE>.

Argument *options* evaluates as a string and is used as set of special flags case insensitive. Values of *options* flags see in special chapter *Regular Expressions*, section *Regular Expressions Options*. Regular expressions syntax is described in chapter *Regular Expressions*, section *Regular Expressions Syntax*.

Examples:

```
USER>w $zpcrematch("one two","(\w+) (\w+)")
1
USER>w $zpcrematch("one two three","(\w+) (\w+)")
0
```

6.35 $ZPCREREPLACE

Function replace substrings matched to regular expression.

Syntax

$ZPCREREPLACE(str,regexp,with[,options])

$ZPCRER(str,regexp,with[,options])

Definition

str	Source string to replace substrings in.
regexp	Regular expression.
with	String to substitute.
options	Matching options, set of special flags, optional.

Function *$zpcrereplace()* search and replace in string *str*, substrings matched

to regular expression *regexp* to string *with* and return result. If argument *options* is present and contains matching flag "G", function replace all matched occurences, otherwise replace only first found.

If argument *options* is present and contains matching flag "P", function use value of *with* as string with pseudovariables. If pseudovariables has been found in *with*, it substitutes.

Pseudovariables in *with* starts with "$" symbol and used the following escape rules:

$$		Outputs only one symbol$
$NNN ${NNN}	or	Digits sequence NNN is used as a pseudovariable number

Other symbol sequence after "$" symbol is used as is. Pseudovariables numbers starts from 1. Special symbols { and } allow to specify which of NNN digit is part of pseudovariable number and which not.

Function place into pseudovariable value part of source *str*, matched to regular expression part inside parenthesis.

Argument *options* evaluates as a string and is used as set of special flags case insensitive. Values of *options* flags see in special chapter *Regular Expressions*, section *Regular Expressions Options*. Regular expressions syntax is described in chapter *Regular Expressions*, section *Regular Expressions Syntax*.

Examples:

Single replace with pseudovariables:

```
USER>s re="(\w+) (\w+)"

USER>s str="**one two##three four"

USER>w $zpcrer(str,re,"-=$2 + $1=-","p")
**-=two + one=-##three four
```

Multiple replace with pseudovariables:

```
USER>s re="(\w+) (\w+)"

USER>s str="**one two##three four+++"
```

```
USER>w $zpcrer(str,re,"-=${2}22 + $1=-","pg")
**-=two22 + one=-##-=four22 + three=-+++
```

Multiple replace without pseudovariables:

```
USER>s re="(\w+) (\w+)"

USER>s str="**one two##three four+++"

USER>w $zpcrer(str,re,"-=${2}22 + $1=-","g")
**-=${2}22 + $1=-##-=${2}22 + $1=-+++
```

If function detect *regexp* has invalid regular expression syntax, function generate an error <PCRE>. If function cannot complete replace using varible maximum length limitation (32K), function generate an error <MAXSTRING>.

6.36 $ZPCRESEARCH

Function search substrings matched to regular expression.

Syntax

$ZPCRESEARCH(str,regexp[,options])

$ZPCRES(str,regexp[,options])

Definition

str	Expression evaluated as a string to search substrings in.
regexp	Regular expression.
options	Matching options, set of special flags, optional.

Function *$zpcresearch()* search in string *str* one or more substrings specified by regular expression. If argument *options* is present and contain symbol "G", function search all occurences, otherwise search only first matched substring.

If function detect *regexp* has invalid regular expression syntax, function generate an error <PCRE>.

Function *$zpcresearch()* returns result as list of substrings found, in *$list-*

build() function format. If no any mathes found, function return an empty string (valid list with no elements). Result contains substrings in the same order has been found.

Argument *options* evaluates as a string and is used as set of special flags case insensitive. Values of *options* flags see in special chapter *Regular Expressions*, section *Regular Expressions Options*. Regular expressions syntax is described in chapter *Regular Expressions*, section *Regular Expressions Syntax*.

Examples:

```
USER>s re="(\w+) (\w+)"

USER>s str="one two three four five six seven"

USER>s found=$zpcresearch(str,re,"g")

USER>f i=1:1:$ll(found) w $lg(found,i),!
one two
three four
five six
```

6.37 $ZPOWER

Function evaluates a number raised to specified power.

Syntax

$ZPOWER(Number,Exponent)

Definition

Number	Expression evaluated as a number to raise.
Exponent	Expression evaluated as a number to use as exponent.

Function evaluates a *Number* raised to *Exponent* power.

Function *$zpower()* performs tha same action as an exponentaion operator (**).

```
$zpower(x,y)=x**y
```

If *Number* evaluates equal 0, the *Exponent* value must be positive number.

If *Number* is less than 0, the *Exponent* value must be an integer.

Examples:

```
TEMP>w $zpower(2,10)
1024
TEMP>w $zpower(2,-0.5)
.707106781186548
```

6.38 $ZQASCII

Function converts a eight-byte substring of string to a number.

Syntax

$ZQA[SCII](String[,Position])

Definition

String	Expression evaluated as a string to get eight-byte substring from.
Position	Expression evaluated as an integer to get eight-byte substring from. Optional.

Function *$zqascii()* return integer calculated from eight-byte substring from source *String*. Decimal value of a result is converted from ASCII codes of bytes of specified substring. If argument *Position* does not specified, function gets first eight-byte substring. If source *String* is an empty string, function return value -1. If from specified *Position* source *String* does not contain eight bytes, function return value -1.

Function *$zqascii()* can be represented using standard functions and operators by formula:

$zqa(str,pos) = $a(str,pos) + ($a(str,pos+1) * 256) + ($a(str,pos+2) * 256 * 256 * 256) + ($a(str,pos+3) * 256 * 256 * 256) + ($a(str,pos+4) * 256 * 256 * 256 * 256 * 256) + ($a(str,pos+5) * 256 * 256 * 256 * 256 * 256 * 256) + ($a(str,pos+6) * 256 * 256 * 256 * 256 * 256 * 256 * 256) + ($a(str,pos+7) * 256 * 256 * 256 * 256 * 256 * 256 * 256 * 256)

Examples:

```
TEMP>w $zqa("123456")
-1
```

```
TEMP>w $zqa("123456789")
4050765991979987505
TEMP>w $zqa("123456789",2)
4123106164818064178
TEMP>w $zqa("123456789",-5)
-1
TEMP>w $zqa("123456789",5)
-1
```

Notes: Internal MiniM integers are represented by 32-bit or 64-bit signed integers, and some of numbers can be displayed as negative even if functions *$zqa()* and *$zqc()* still work with him as with positive numbers, using as unsigned bytes sequence.

```
USER>w $zqa("internet")
8387231318654873193
USER>w $zqc($zqa("internet"))
internet
```

For arithmetic operations maximum integer is:

```
TEMP>w 256*256*256*256*256*256*256*127
9151314442816847872
```

6.39 $ZQCHAR

Convert integer number to eight-byte string.

Syntax

$ZQC[HAR](Integer)

Definition

Integer	Expression evaluated as an integera value to convert to eight-byte string.

Function *$zqchar()* convert integer number to eight-byte string. Function use ASCII codes to represent symbols in string. Negative numbers are used as a complement to full 64-bit unsigned integer. Function *$zqchar()*, unlike *$char()*, *$zwchar()*, and *$zlchar()* functions, does not return an empty string.

Function *$zqchar()* can be represented using standard functions and operators by formula:

```
$zqc(n) = $c(n#256,n\256#256,n\(256**2)#256,n\(256**3),
    n\(256**4),n\(256**5),n\(256**6),n\(256**7))
```

Examples:

```
TEMP>zzdump $zqc(1234567890123)
0000: CB 04 FB 71 1F 01 00 00
USER>zzdump $zqc(-1)
0000: FF FF FF FF FF FF FF FF
```

Notes: Internal MiniM integers are represented by 32-bit or 64-bit signed integers, and some of numbers can be displayed as negative even if functions *$zqa()* and *$zqc()* still work with him as with positive numbers, using as unsigned bytes sequence.

```
USER>w $zqa("internet")
8387231318654873193
USER>w $zqc($zqa("internet"))
internet
```

For arithmetic operations maximum integer is:

```
TEMP>w 256*256*256*256*256*256*256*127
9151314442816847872
```

6.40 $ZQUOTE

Return string with text decorated as valid MUMPS expression.

Syntax

$ZQUOTE(String)

Definition

String	Expression evaluated as a string to decorate.

Function *$zquote()* return a string with valid MUMPS text representation

of argument. Result is returned as MUMPS code expression ready to use in MUMPS routines. Function give a proper weigh of double quote symbol, nonprintable symbols and *$char()* function limitation by argument count. Result is created using *$char()* function, numbers, concatenation operator and quoted strings id need. Ordinal numbers which does not require text decoration returnes as is. All symbols with codes greater than 127 are used as a printable symbols.

Examples:

```
USER>w $zquote("internet")
"internet"
USER>w $zquote("internet"_$c(1,2,3)_""" quote")
"internet"_$C(1,2,3)_""" quote"
USER>w $zquote(789456)
789456
USER>w $zquote($zlc(789456))
"?"_$C(11,12,0)
USER>zzdump $zquote($zlc(789456))
0000: 22 D0 22 5F 24 43 28 31 31 2C 31 32 2C 30 29
```

6.41 $ZSEC

Function returns the trigonometric secant of the specified angle value in radians.

Syntax

$ZSEC(Angle)

Definition

Angle	Expression evaluated as a number and used as an angle in radians.

Function evaluates ans returns the trigonometric secant of the specified *Angle* in radians.

Function as all other trigonometric functions use values rounded to available decimal digits and calculations can have some miscalculation.

Examples:

```
TEMP>w $zsec($zpi/4)
```

```
1.41421356237309
```

6.42 $ZSIN

Function returns the trigonometric sine of the specified angle value in radians.

Sintax

$ZSIN(Angle)

Definition

Angle	Expression evaluated as a number and used as an angle in radians.

Function evaluates and returns the trigonometric sine of the specified *Angle* in radians.

Examples:

```
>w $zsin(-3*$zpi/4)
-.7071067811865475242
```

Function as all other trigonometric functions use values rounded to available decimal digits and calculations can have some miscalculation.

```
TEMP>w $zsin($zpi)
1.22460635382238E-16
```

6.43 $ZSQR

Function returns the square root of a specified number.

Syntax

$ZSQR(Number)

Definition

Number	Expression evaluated as a number.

Function *$zsqr()* evaluates and returns square root of a *Number*.

If an argument *Number* evaluates as a negative number, functin generate

an error <ILLEGAL VALUE>.

Examples:

```
TEMP>w $zsqr(4)
2
TEMP>w $zsqr(-4)

<ILLEGAL VALUE>
TEMP>w $zpower(2,0.5)=$zsqr(2)
1
```

6.44 $ZTAN

Function returns the trigonometric tangent of the specified angle value in radians.

Syntax

$ZTAN(Angle)

Definition

Angle	Expression evaluated as a number and used as an angle in radians.

Function *$ztan()* evaluates and returns the trigonometric tangent of the specified *Angle* in radians.

Function as such as all other trigonometric functions use values rounded to available decimal digits and calculations can have some miscalculation.

Examples:

```
TEMP>w $ztan($zpi/2)
1.63317787283838E+16
TEMP>w $ztan($zpi/4)
1
```

6.45 $ZTIME

Function converts time from *$horolog* format to a specified by parameter.

Syntax

$ZT[IME](Value[,Format])

Definition

Value	Expression evaluated as an integer, time in *$horolog* second fraction, seconds since day begin.
Format	Optional, expression evaluates as a number, used as integer format code. If not specified, by default used value 1.

Value of *Value* can be from 0 to 86400, and if *Value* evaluates outside of this interval, function generate an error <ILLEGAL VALUE>.

Value of argument *Format* specify one of the following supported formats:

1	HH:MM:SS	24 hours
2	HH:MM	24 hours
3	HH:MM:SS(AM/PM)	12 hours
4	HH:MM(AM/PM)	12 hours

Examples:

```
USER>w $zt(12345,1)
03:25:45
USER>w $zt(12345,2)
03:25
USER>w $zt(12345,3)
03:25:45AM
USER>w $zt(12345,4)
03:25AM
```

For other values of *Format* function generate an error <FUNCTION>.

Delimiters are used by function *$ztime()* does not depend of MiniM service account. Hours, minutes and seconds are always formatted with leading zeroes.

To use *$horolog* with *$ztime()* function may be used function *$piece()*, for example:

```
w $ztime($piece($horolog,",",2))
```

6.46 $ZTIMEH

Function converts time from string-formatted representation to an integer in *$horolog* format, second fraction.

Syntax

$ZTIMEH(Value[,Format])

$ZTH(Value[,Format])

Definition

Value	Expression with string representation of the time to convert from.
Format	Expression with format code. Optional, by default used code 1.

If *Value* contains impossible hours, minutes or seconds, function generate an error <ILLEGAL VALUE>.

Value of argument *Format* specify one of the following supported formats:

1	HH:MM:SS	24 hours
2	HH:MM	24 hours
3	HH:MM:SS(AM/PM)	12 hours
4	HH:MM(AM/PM)	12 hours

For other values of *Format* function generate an error <FUNCTION>.

Delimiters are used by function *$ztime()* does not depend of MiniM service account. Hours, minutes and seconds cen be specified with leading zeroes.

6.47 $ZUCASE

Function convert all symbols to upper case.

Syntax

$ZUCASE(String)

Definition

String	Expression evaluated as a string to convert from.

Function *$zucase()* return string based on argument with converting all symbols to upper case. Conversion is made using current locale file. See configuration file minim.ini, section Server, key Locale.

Examples:

```
TEMP>w $zucase("MiniM")
MINIM
```

6.48 $ZUPPER

Function convert all symbols to upper case.

Syntax

$ZUPPER(String)

Definition

String	Expression evaluated as a string to convert from.

Function *$zupper()* return string based on argument with converting all symbols to upper case. Conversion is made using current locale file. See configuration file minim.ini, section Server, key Locale.

Examples:

```
TEMP>w $zupper("MiniM")
MINIM
```

6.49 $ZVERSION

Function returns separate parts of the product version.

Syntax

$ZV[ERSION](Part)

Definition

Part	Number of the product version part.

Function returns separate parts of the product version - bits, target op-

erating system, product version, product name or other.

Return values:

Part	Value
0	Number of product version, for example 1.15
1	Product name, for example MiniM or MiniMono or other.
2	Target operating system for which this product was built, for example Windows or Linux or other.
3	Product bits - 32, 64 or other.
4	Product build type - release, debug, test, or other.
5	Target processor architecture - i386, x64 or other.

For example, in the MiniM for Windows in 32 - bit edition function $zversion returns the following:

```
USER>w $zv(0)
1.15
USER>w $zv(1)
MiniM
USER>w $zv(2)
Windows
USER>w $zv(3)
32
```

6.50 $ZWASCII

Function converts a two-byte substring of string to a number.

Syntax

$ZWA[SCII](String[,Position])

Definition

String	Expression evaluated as a string to convert two-byte substring from.
Position	Expression evaluated as an integer, position of two-byte substring.

Function *$zwascii()* extract two-byte substring of string and converts into integer using ASCII codes of bytes. If argument *Position* is not specified, function use first two bytes of *String*. If *String* is an empty string, function return value -1. And function return value -1 if *Position* evaluates less than 0 or from specified position *String* does not contain both two bytes.

Function *$zwascii()* can be represented using standard functions and operators by formula:

$zwa(str,pos) = $a(str,pos) + ($a(str,pos+1) * 256)

Examples:

```
TEMP>w $zwa("")
-1
TEMP>w $zwa("123456")
12849
TEMP>w $a("1")+($a(2)*256)
12849
TEMP>w $zwa("123456",4)
13620
TEMP>w $a("4")+($a("5")*256)
13620
TEMP>w $zwa("123456",-1)
-1
TEMP>w $zwa("123456",-5)
-1
TEMP>w $zwa("123456",8)
-1
```

6.51 $ZWCHAR

Function converts a number to a two-byte string.

Syntax

$ZWC[HAR](Integer)

Definition

Integer	Expression evaluated as an integer to convert two-byte string from.

Function *$zwchar()* return two-byte string converted from *Integer* using ASCII codes. If *Integer* id less than 0 or greater than $256 * 256 = 65536$, function return an empty string.

Function *$zwchar()* can be represented using standard functions and operators by formula:

```
$zwc(n) = $c(n#256,n\256)
```

Examples:

```
TEMP>w $zwc(25185)
ab
TEMP>w $zwc(2518500)

TEMP>w $zwc(-456)

TEMP>
TEMP>w $zwc(123)
{
TEMP>zzdump $zwc(123)
0000: 7B 00
```

Chapter 7

System Variables

7.1 $DEVICE

System variable for state of current device.

Syntax

$DEVICE

$D

Definition

System variable *$device* is implemented for state of current device storage. On process start value of *$device* is an empty string.

MiniM Database Server now does not use or change this variable on any side effect and variable is intended for programmers usage by standard. Recommended format is a string comma delimited:

```
stderr,usererr,explanation
```

stderr	Input-output error code.
usererr	User error code.
explanation	Error's text description.

Variable *$device* can be assigned by the *set* command

Examples:

```
TEMP>w $d
```

```
TEMP>s $d=123456

TEMP>w $d
123456
```

7.2 $HOROLOG

Returns current date and time in local time comma delimited.

Syntax

$HOROLOG

$H

Definition

System variable *$horolog* return current date and time in local time comma delimited. Format is:

```
Number of days since Dec 31, 1840
Comma
Number of seconds since day starts.
```

First day is Jan 1, 1841. Format of *$horolog* variable is a MUMPS standard format and is used by most MUMPS programs. Example:

```
TEMP>w $h
60987,56260
```

To get day number and second number there can be used a *$piece()* function, for example:

```
TEMP>s h=$h w $p(h,",",1),!,$p(h,",",2)
60987
56348
```

System variable *$horolog* format is used by all other system funtions to handle days and time: *$zd()*, *$zt()*, *$zdh()*, *$zth()* and closest format is used by system variable *$ztimestamp*.

Value of system variable *$horolog* evaluates each call to variable and two serial call to variable can return different values. To handle date and time

togethe properly it is required get pair of day and time only once and handle this pair. Otherwise it is possible two calls to *$horolog* can be made in different days.

To use only day from full date and time there is possible to use unary plus operator to transform *$horolog* format to day only:

```
TEMP>w $h
60987,56598
TEMP>w +$h
60987
```

7.3 $ECODE

Returns the current error code string.

Syntax

$ECODE

$EC

Definition

On read the system variable *$ecode* returns current error code string with last occured errors delimited by comma. Error codes included into MUMPS standard, are prefixed by the "M" symbol. Error codes added by MiniM are prefixed by the "Z" symbol. Standard recommended to use prefix "U" for the user-defined errors. Other prefixes are reserved by MUMPS standard for future use.

If occurs an error defined by MUMPS standard, MiniM adds symbol "M" with following error code.

Examples:

```
TEMP>s $ec=""

TEMP>w a

<UNDEFINED>
TEMP>w b

<UNDEFINED>
TEMP>w $ec,!,$ze
```

```
,M6,M6,
<UNDEFINED>
```

If occurs an error not included into MUMPS standard and intended by MiniM, MiniM adds symbol "Z" with following error string. For example, if runs cycle and user press Ctrl+C:

```
TEMP>s $ec=""

TEMP>f

<INTERRUPT>
TEMP>w $ec,!,$ze
,Z<INTERRUPT>,
<INTERRUPT>
```

Value of *$ecode* variable can be assigned by the *set* command. If this value is an empty string, assignment done and have no other side effect. Otherwise value of *$ecode* is assigned, value of *$zerror* changes to "<ECODETRAP>" and process generate an error <ECODETRAP>.

Examples:

```
TEMP>s $ec="abcd"

<ECODETRAP>
TEMP>w $ec,!,$ze
abcd
<ECODETRAP>
TEMP>s $ec=""

TEMP>w $ec,!,$ze

<ECODETRAP>
```

System variable *$ecode* is present only in single instance and does not stacked on stack level creation.

7.4 $ESTACK

Returns current stack level relative last user-defined point.

Syntax

$ESTACK

$ES

Definition

System variable *$estack* returns value of current stack level counted from last user-defined stack save point. On process start this variable have value 0. On each stack level creation value of *$estack* automatically increments by one.

Unlike of system variable *$stack* with automatically decrements by one on stack level leaving, variable *$estack* on stack level leaving restored to previous value was on previous stack level.

Value of *$estack* can be zeroed by the *new* command:

```
new $estack
```

On this code execution value of *$estack* sets to 0 and on return to this stack level value of *$estack* is restored to 0 too. Values of *$estack* on previous stack levels are stored unchanged and can be nonzero.

Value of *$estack* cannot be assigned by the *set* command.

Unlike of system variable *$stack* value of *$estack* can be restored from zero to previous nonzero value on stack leaving.

7.5 $ETRAP

Contains string of commands to be executed when an error occurs.

Syntax

$ETRAP

$ET

Definition

System variable *$etrap* contains string of commands to be executed when an error occurs.

When process starts, value of *$etrap* is an empty string.

Value of *$etrap* can be assigned by the *set* command. On direct assignment unlike *$ztrap* assignment MiniM does not check syntax of new *$etrap* value.

System variable *$etrap* can have separate value on each stack level. When new stack level creates, new value of *$etrap* is copied from previous stack level. When new separate value was created, all assignments change this value and does not change all previous.

Value of *$etrap* can be created as separate value on current stack level by the *new* command. If *new* command applies to the *$etrap* variable, process creates new *$etrap* value and copies value from previous stack level. If was used *new* command with assignment, process creates new *$etrap* value and copies value from specified expression.

Examples:

```
TEMP>s $et="w !123,!"

TEMP>n $et

TEMP>w $et
w !123,!
TEMP>n $et="789"

TEMP>w $et
789
TEMP>
```

When process execution leave stack level, value of *$etrap* is automatically restored to previous value.

When error was occured, value of *$etrap* executes as command sequence instead of current execution line of code.

7.6 $IO

Return device identification string of current device.

Syntax

$IO

$I

Definition

System variable *$io* return device identification string of current input-output device. After current device changed variable *$io* changes the value too unlike of *$principal* variable.

Current device can be changed by the *use* command or by the *close* command with closing current device (MiniM automatically make current a principal device) and if process returns into interactive mode in console or in telnet.

Examples:

```
TEMP>s dev="|NULL|123" o dev u dev s d=$io s p=$p c dev

TEMP>w
d="|NULL|123"
dev="|NULL|123"
p="|CON|"

TEMP>w $i
|CON|
```

7.7 $JOB

Returns current job number.

Syntax

$J[OB]

Definition

System variable *$job* returns number of current process executed. This value still unchanged all time until process terminates. At the same time all MiniM processes have different numbers, but after termination next MiniM process can get this used job number. Job number have not predefined interval and is operating-system dependent.

Example:

```
TEMP>w $j
1608
```

MiniMono difference

MiniM Database Server in a $job value always return the Windows process number, and MiniM Embedded Edition always return the Windows thread number.

7.8 $KEY

Returns last read termination.

Syntax

$KEY

$K

Definition

System variable *$key* contains symbols sequence that terminated last *read* command from current device. For each device MiniM supports owned value of *$key* variable and value of *$key* can be changed if current device is changed.

If process work in interactive mode (console or telnet), value of *$key* variable can be owerwritten while operator input commands. Ordinary read comands in command input mode terminates by Carriage Return and it is a start value of *$key* for commands executed in interactive mode.

Example:

```
TEMP>r s s k=$k w ! zzdump k
ww
0000: 0D
```

If *read* command terminates by terminator read, variable *$key* contains value of terminator.

If *read* command terminates by timeout expiration, variable *$key* contains an empty string.

On *read* one symbol (*read *ch*) variable *$key* contains only this symbol was read. This behavior does not specified by MUMPS standard and is a MiniM extended behavior.

Value of variable *$key* can be assigned by the *set* command. Assigned value is stored in internal device data and saved until next *read* command executed for this device.

7.9 $PRINCIPAL

Returns device identification string of principal device for current process.

Syntax

$PRINCIPAL

$P

Definition

System variable *$principal* returns device identification string of principal device of current process. This device creates automatically on process start and cannot be closed. Value of variable *$principal* still unchanged until process terminates. This device automatically still current on process execution return to top level in console or telnet mode and on closing current device. Closing of principal device ignored by MiniM and have no any side effects or error generation.

Examples:

```
TEMP>s dev="|NULL|123" o dev u dev s d=$io s p=$p c dev

TEMP>w
d="|NULL|123"
dev="|NULL|123"
p="|CON|"

TEMP>w $i
|CON|
```

7.10 $QUIT

Returns flag indicating quit with argumets is required in current context.

Syntax

$QUIT

$Q

Definition

System variable *$quit* return glag indicating *quit* command with or without argument is required in current execution context. If context requires

return value, *$quit* returns value 1, otherwise 0. Return is required is current execution context has been created by $$ call and does not required for *do* and *xecute* context.

If *quit* with argument is required, the *quit* command without argument generate an error and vice versa.

System variable *$quit* can be used to determine what *quit* command form need to be executed for functions which can be called in both contexts - $$ call and *do* call.

7.11 $REFERENCE

Returns last global reference.

Syntax

$REFERENCE

$R

Definition

System variable *$reference* contains last global reference process made. This value also have name naked indicator value or naked reference. On process start last global reference is not defined and value of *$reference* variable is an empty string.

Naked indicator changes on global variable evaluation, assignment, killing and using functions *$data()*, *$order()*, *$query()*, *$increment()* if functions call global variable.

System variable *$reference* can be changed by assignment using the *set* command without call global variables. Variable can be assigned to any valid global name (existing or not) and empty string. On assignment MiniM check syntax and if value is not valid global name, process generate an error <SYNTAX>. For example:

```
TEMP>s $r="^aa"

TEMP>w $r
^aa
TEMP>s $r="^bb(123)"

TEMP>w $r
```

```
^bb(123)
TEMP>s $r=""

TEMP>w $r

TEMP>
```

If one of functions *$data()*, *$order()*, *$query()* or *$increment()* call global variable with database name specification, naked reference store database name too, otherwise not. Examples:

```
TEMP>w $d(^|"TEMP"|aa)
0
TEMP>w $r
^|"TEMP"|aa
TEMP>w $d(^aa)
0
TEMP>w $r
^aa
TEMP>
```

If process changes current database, value of named indicator automatically clears. If new database is the same as old, nothing to do and naked indicator does not changes.

Function *$data()* always changes value of naked indicator, even if specified global name does not contain data or subscripts with data and function returns 0.

Functions *$order()* and *$query()* if next global variable exists, changes value of naked indicator to next found global name, otherwise changes naked indicator to start global name.

Function *$increment()* with global name, if executes successfully, always changes value of naked indicator.

7.12 $STACK

Returns current execution stack level.

Syntax

$STACK

$ST

Definition

System variable *$stack* returns number of stack level for current execution context. Stack levels counts from 0 (top level). Each call of $$ function, *do* procedure or *xecute* line of code automatically creates new stack level and value of *$stack* is incremented by one. On stack level leaving value of *$stack* is decremented by one.

Value of system variable *$stack* cannot be assigned by the *set* comand.

Unlike of system variable *$estack*, system variable *$stack* cannot be zeroed by the *new* command.

7.13 $STORAGE

Returns value of available local storage in bytes.

Syntax

$STORAGE

$S

Definition

System variable *$storage* returns value of local storage for current process in bytes. It is memory size available to store local variables. Initial value of *$storage* is defined in configuration file minim.ini, section Process, key Storage.

Variable returns total free bytes for local storage, but this value is not precise to determine can be local variable stored or not, because internal allocation algorithm allow small fragmentation of memory. And not all free memory size can be used to store variables.

MiniM use local storage area to store local variables values, local variables names and names synonims created by passing local variables by reference.

Value of system variable *$storage* cannot be changed by the *set* command.

7.14 $SYSTEM

Returns string with current MiniM instance identification.

Syntax

$SY[STEM]

Definition

System variable *$system* returns string with current MiniM instance identification. String consists with current computer name and MiniM instance name delimited by colon. Instance name is specified in configuration file minim.ini and still unchanged until current instance is run. If on the same computer was installed several MiniM instances, it is required to have different instance names.

Examples:

```
TEMP>w $system
AUGUST:MINIMOO
TEMP>w $sy
AUGUST:MINIMOO
```

MiniMono difference

MiniM Embedded Edition always use instance name "MINIMONO" and $system variable returns computer name and "MINIMONO" string.

7.15 $TEST

Returns the truth value resulting from the last *if* command or command using the time-out option.

Syntax

$TEST

$T

Definition

System variable *$test* returns current the truth value resulting from the last *if* command or command using the time-out option.

The *read* command with timeout sets value of the *$test* variable into 1 if reading was ended before timeout expired. If *read* terminates by timeout expiration, the *$test* variable sets into 0. The *read* command without timeout does not change value of the *$test* variable.

The *lock* command with timeout sets value of the *$test* variable into 1 if locking was ended before timeout expired. If *lock* terminates by timeout

expiration, the *$test* variable sets into 0. The *lock* command without timeout does not change value of the *$test* variable.

The *open* command with timeout sets value of the *$test* variable into 1 if opening was ended before timeout expired. If *open* terminates by timeout expiration, the *$test* variable sets into 0. The *open* command without timeout does not change value of the *$test* variable.

The *open* command sets value of the *$test* variable into 1 if child process was run successfully, otherwise sets to 0. The *job* command without timeout does not change value of the *$test* variable.

The *use* command for TCP device with /ATO option sets the *$test* variable into 1 if external connection has been made within timeout. This is MiniM additional behavior and does not inluded into MUMPS standard.

The *if* command with argument sets value of the *$test* variable into 1 if expression of argument evaluates as nonzero, otherwise sets to 0. Argumentless form of the *if* command check the value of current *$test* value and does not change value of *$test*.

Value of the *$test* variable is used by the argumentless *if* command and by *else* command. The *else* command continues execution of following commands in line if value of *$test* is 0.

System variable *$test* cannot be changed otherwise or assigned by the *set* command.

Commands without timeouts does not change value of the *$test* variable. And, value of the *$test* variable does not changed by postconditional expression evaliating.

Values of the *$test* variable are stacked and protected by changing if user function is called ($$ context) and in context of argumentless *do* command. In context of *xecute* command execution and in context of argumented *do* command value of the *$test* variable is not stacked and does not protected, can be changed at the next stack level. Examples how works stack protection of system variable *$test* using routine::

```
run()
 n tmp
 w "argumentless do",!
 i 1 w $t,! d  w $t,!
 . i 0
 w "do with argument",!
 i 1 w $t,! d proc w $t,!
```

```
 w "xecute",!
 i 1 w $t,! x "i 0" w $t,!
 w "function",!
 i 1 w $t,! s tmp=$$func() w $t,!
 q
proc
 i 0
 q
func()
 i 0
 q 1
```

On execution we got result:

```
argumentless do
1
1
do with argument
1
0
xecute
1
0
function
1
1
```

On process start value of *$test* is 0.

Value of the *$test* variable cannot be other than 0 or 1.

7.16 $TLEVEL

Returns current transaction level.

Syntax

$TLEVEL

$TL

Definition

System variable *$tlevel* returns value of current transaction context. This variable cannot be assigned by the *set* command. Value of variable increments by one on each *tstart* command, decrements by one on each *tcommit* command and sets to 0 on *trollback* command. On process start transaction level is 0 and process is outside of transaction.

Examples:

```
TEMP>w $tl,! ts  w $tl,! ts  w $tl,! tc  w $tl,! tc  w $tl,!
0
1
2
1
0

TEMP>w $tl,! ts  ts  w $tl,! tro  w $tl,!
0
2
0
```

7.17 $X

Returns or changes horizontal position of the caret.

Syntax

$X

Definition

System variable *$x* returns or changes with assignment horizontal position of caret.

This system variable applies only to current device and each device have owned *$x* value.

System variable *$x* by default is terminal-oriented and intended to support console and telnet devices. For other device types this value is type-dependent and in most cases return 0 value.

System variable *$x* counts from 0 and increase from left to right.

Console device (|CON|)

On evaluating value of the *$x* variable process calls Windows console.

On assignment value of the *$x* variable process calls Windows console too to change current caret position.

Telnet device (|TNT|)

On evaluating value of the *$x* variable process returns internal counted value. Value counts on *read*, *write* and *zwrite* commands execution.

On assignment value of the *$x* variable device sends special telnet escape sequence to client and changes internal *$x* value. MiniM suppose than telnet client change current caret position correctly.

Caution! When executed command

```
write *char
```

devices does not evaluate current *$x* position changes and after this command process does not guarantee *$x* value correctness.

7.18 $Y

Returns or changes vertical position of the caret.

Syntax

$Y

Definition

System variable *$y* returns or changes with assignment vertical position of caret.

This system variable applies only to current device and each device have owned *$y* value.

System variable *$y* by default is terminal-oriented and intended to support console and telnet devices. For other device types this value is type-dependent and in most cases return 0 value.

System variable *$y* counts from 0 and increase from left to right.

Console device (|CON|)

On evaluating value of the *$y* variable process calls Windows console.

On assignment value of the *$y* variable process calls Windows console too to change current caret position.

Telnet device (|TNT|)

On evaluating value of the *$y* variable process returns internal counted value. Value counts on *read*, *write* and *zwrite* commands execution.

On assignment value of the *$y* variable device sends special telnet escape sequence to client and changes internal *$y* value. MiniM suppose than telnet client change current caret position correctly.

Caution! When executed command

```
write *char
```

devices does not evaluate current *$y* position changes and after this command process does not guarantee *$y* value correctness.

Chapter 8

System Z - Variables

8.1 $ZCHILD

Returns number of last created by *job* command child process.

Syntax

$ZCHILD

Definition

System variable *$zchild* returns number of last child process created by the *job* command. This value of variable still unchanged until next child process is created even if child process terminates. If the *job* command failed to create child process, variable *$zchild* returns an empty string.

Example:

```
s childexist=$d(^$JOB($zchild))
```

8.2 $ZEOF

Returns indicator end of input data.

Syntax

$ZEOF

Definition

System variable *$zeof* returns indicator end of input data reached on read data from device. If current device id a device with types FILE, PIPE, TCP,

STORE or ZDEVICE implemented, this system variable returns value 1 if end of data reached or 0.

If device does not support end-of-file detection, variable *$zeof* always returns value 0.

Indicator is active if end-of-file detection mode is on for current device. By default end-of-file detection mode is off end if device detect end-of-file of input data, device generate an error <ENDOFFILE>.

End-of-file detection can be enabled in configuration file minim.ini, section Process, key TrapOnEof. If this value is 1, process generate an error <ENDOFFILE>, if value is 0, error is not generated and *$zeof* return indicator.

Current end-of-file detection mode can be changed using function *$view("proc",5)*.

8.3 $ZERROR

Returns text of last error occured.

Syntax

$ZE[RROR]

Definition

System variable *$zerror* returns text of last error occured with possible additional information about place or other error details. Error text starts with error name embraced with angle brackets. For example:

```
TEMP>w abc

<UNDEFINED>
TEMP>w $ze
<UNDEFINED>
```

Full supported errors list is listed in current document, chapter Errors List.

Unlike of system variable *$ecode*, value or system variable *$zerror* have no standard meaning and return not last errors list, but only one last error occured in explaned format with short name and possible additional information.

System variable *$zerror* may be assigned by the *set* command. For example:

```
TEMP>w abc

<UNDEFINED>
TEMP>w $ze
<UNDEFINED>
TEMP>s $ze=""

TEMP>w $ze

TEMP>s $ze="789"

TEMP>w $ze
789
```

If system variable *$zerror* have been assigned by the *set* command, MiniM does not check value assigned.

8.4 $ZGUID

Generate and returns GUID (global unique identifier).

Syntax

$ZGUID

Definition

On each call system variable *$zguid* generate new and return global unique identifier. Return value is a GUID in the most common used format in standard hexadecimal representation. Uniqueness of new GUID value guaranteed by Windows operating system.

In the Linux version MiniM uses pseudo-random number generator to generate GUID value.

In the FreeBSD version MiniM uses FreeBSD builtin GUID generator to generate GUID value.

Example:

```
TEMP>w $zguid
C4DC06D9-F40D-4445-ADE6-BABC87DE25C1
```

8.5 $ZHOROLOG

Returns second elapsed from MiniM server start.

Syntax

$ZHOROLOG $ZH

Definition

System variable *$zhorolog* returns seconds elapsed from MiniM server start. Seconds is returned with microseconds precision. MiniM use Windows high resolution timer to measure time elapsed. This variable precision is enough to measure execution time of any command.

Examples:

```
USER>s a=$zh s b=123 s c=$zh

USER>w
a="20.931194"
b=123
c="20.931259"
```

8.6 $ZNAME

Returns currently executed routine name.

Syntax

$ZNAME

Definition

System variable *$zname* returns name of currently executed routine or an empty string if current routine context does not exists. This system variable return is equal of function *$t(+0)* return. If code is executed on top level in console or telnet mode, it have not routine context and variable *$zname* returns an empty string.

Example:

```
s curroutine=$zname
```

8.7 $ZNSPACE

Returns current database name.

Syntax

$ZNSPACE

Definition

System variable *$znspace* returns name of current database name. Database name returned in upper case. Value of *$znspace* variable automatically changes on current database changing and cannot be assigned by the *set* command.

Example:

```
TEMP>w $znspace
TEMP
```

8.8 $ZPARENT

Returns number of parent process, who did run this process.

Syntax

$ZPARENT

Definition

System variable *$zparent* returns number of process who run this process. If current process was run not by *job* command, variable returns an empty string. Parent job number still unchanged all time even if parent process terminates.

Examples:

```
s parentexist=$d(^$JOB($zparent))
```

MiniMono difference

MiniM Embedded Edition in the $zparent value always returns 0.

8.9 $ZPI

Returns value of π constant.

Syntax

$ZPI

Definition

System variable *$zpi* returns value of π constant.

Example:

```
TEMP>w $zpi
3.14159265358979
```

8.10 $ZREFERENCE

Returns last global reference.

Syntax

$ZREFERENCE

$ZR

Definition

System variable *$zreference* contains last global reference process made. This value also have name naked indicator value or naked reference. On process start last global reference is not defined and value of *$zreference* variable is an empty string.

Naked indicator changes on global variable evaluation, assignment, killing and using functions *$data()*, *$order()*, *$query()*, *$increment()* if functions call global variable.

System variable *$zreference* can be changed by assignment using the *set* command without call global variables. Variable can be assigned to any valid global name (existing or not) and empty string. On assignment MiniM check syntax and if value is not valid global name, process generate an error <SYNTAX>. For example:

```
TEMP>s $zr="^aa"

TEMP>w $zr
```

```
^aa
TEMP>s $zr="^bb(123)"

TEMP>w $zr
^bb(123)
TEMP>s $zr=""

TEMP>w $zr

TEMP>
```

If one of functions *$data()*, *$order()*, *$query()* or *$increment()* call global variable with database name specification, naked reference store database name too, otherwise not. Examples:

```
TEMP>w $d(^|"TEMP"|aa)
0
TEMP>w $zr
^|"TEMP"|aa
TEMP>w $d(^aa)
0
TEMP>w $zr
^aa
TEMP>
```

If process changes current database, value of named indicator automatically clears. If new database is the same as old, nothing to do and naked indicator does not changes.

Function *$data()* always changes value of naked indicator, even if specified global name does not contain data or subscripts with data and function returns 0.

Functions *$order()* and *$query()* if next global variable exists, changes value of naked indicator to next found global name, otherwise changes naked indicator to start global name.

Function *$increment()* with global name, if executes successfully, always changes value of naked indicator.

8.11 $ZTIMESTAMP

Returns current date and time in GMT with milliseconds.

Syntax

$ZTIMESTAMP

$ZTS

Definition

System variable *$ztimestamp* returns current date and time in GMT (unlike of *$horolog* variable) with precision up to milliseconds.

Variable format:

```
number of days since Dec 31, 1840
comma (,)
number of seconds from day start
dot (.)
number of milliseconds from second start
```

On formatting milliseconds MiniM align and pad if need by leading milliseconds zeroes. Leading zeroes are present, and terminating zeroes can be omitted, for example:

Milliseconds	Milliseconds formatted
1	.001
10	.01
100	.1

Caution: every call to this variable can return different values.

Examples:

```
TEMP>w $zts
60986,43395.485
TEMP>w $h
60986,54199
```

8.12 $ZTIMEZONE

Return information about current time zone.

Syntax

$ZTIMEZONE

$ZTZ

Definition

System variable *$ztimezone* returns information about currently used time zone. It is time offset from GMT (Greenwich Mean Time) in minutes. Time zone on the east of GMT are specified as negative numbers. For example, in Moscow time zone is 3 hours or -180 minutes.

Variable *$ztimezone* cannot be changed by the *set* command.

Examples:

```
TEMP>w $ztimezone
-180
TEMP>w $ztz
-180
```

8.13 $ZTRAP

Returns value of last assigned error handler.

Syntax

$ZTRAP $ZT

Definition

System variable *$ztrap* returns value of last assigned extended error handler. This variable have the same value for all stack levels and stores last assigned value.

Error handler assigned using *$ztrap* is not special second error handling mechanism and indended for close as possible compatibility with most widely used MUMPS implementations. This aeeror handler is pseudohandler ans really is only wrapper to standard error handling mechanism. If variable is *$ztrap* assigned, this is equal assigning appropriate values to system variables *$estack* and *$etrap*. Error handler assigned using *$ztrap* is handled by *goto* command. On *$ztrap* MiniM create automatically assignment to variables *$estack* and *$etrap* and if need use *new* command and this action is dependent of new *$ztrap* value.

Value of *$ztrap* must be a label reference to use by *goto* command or label reference prefixed by "*" symbol.

If value of *$ztrap* contains only label reference, this assignment is equal of code execution:

```
new $estack
new $etrap
set $etrap="g:'$es "_$ztrap
```

This conforms to stack unrolling or error until stack level when assignment was made and execution of *goto* command to specified label.

If calue of *$ztrap* contains label reference prefixed by "*" symbol, this assignment is equal of code execution:

```
set $etrap="g "_$e($ztrap,2,$l($ztrap))
```

This conforms execution *goto* command only to specified label independent of stack level when error occurs.

Examples:

```
USER>s $zt="err^errhandler"

USER>w $et
g:'$es err^errhandler
USER>s $zt="*err^errhandler"

USER>w $et
g err^errhandler
```

If system variable *$ztrap* is assigned to any value which MiniM cannot classify as possible, process generate an error <SYNTAX>.

8.14 $ZVERSION

Returns information about currently used MiniM Database Server version.

Syntax

$ZV[ERSION]

Definition

System variable *$zversion* returns string with information about currently used MiniM Database Server version. Version information have not predefined format. Not it includes string "MiniM", version number and build date.

Example:

```
TEMP>w $zv
MiniM for Windows 32 bit ....
```

MiniMono difference

In MiniM Embedded Edition variable $zversion returns string with the MiniMono substring:

```
MiniMono for Windows ...
```

MiniM x64 difference

In MiniM x64 variable $zversion returns string with the architecture implementation of MiniM:

```
MiniM for Windows x64 ...
```

Chapter 9

Structured System Variables

9.1 $DEVICE

Structured system variable ^$DEVICE exposes information about devices opened by current process.

Syntax

^$D[EVICE](deviceid)

Definition

First subscript of variable *deviceid* is a device identification string. Devices are identified by string prefixed by device type with following identification symbols dependent of device type. Commands *open*, *use*, and *close* use device identification strings as an argument.

Variable ^$DEVICE supports only one subscript. If process successfully open device, this device identification string automatically is present in ^$DEVICE.

MiniM Database Server supports ^$DEVICE variable as first argument of functions *$data()*, *$order()* and *$query()*.

^$DEVICE as a *$data()* argument

Function *$data()* with a ^$DEVICE as an argument allow to determine is this device already opened by current process or not. If device is opened and available, dunction return value 1, otherwise 0.

```
TEMP>w $d(^$D($io))
1
TEMP>w $d(^$D("   "))
```

```
0
TEMP>o "|NULL|"

TEMP>w $d(^$D("|NULL|"))
1
```

^$DEVICE as an *$order()* argument

Function *$order()* with a ^$DEVICE as an argument allow to enumerate device identification strings of devices currently opened by process. Function can be used with forward and backward enumerate direction. If function does not detect next device name it returns an empty string.

```
TEMP>s dev="|NULL|" o dev

TEMP>s dev="" f  s dev=$o(^$D(dev)) q:dev=""  w dev,!
|CON|
|NULL|
```

^$DEVICE as a *$query()* argument

Function *$query()* with a ^$DEVICE as an argument allow to enumerate names of ^$DEVICE with device name as first subscript. Function can be used with forward and backward enumerate direction. If function does not detect next device name it returns an empty string.

```
TEMP>s dev="|NULL|" o dev

TEMP>s n=$na(^$d("")) f  s n=$q(@n) q:n=""  w n,!
^$DEVICE("|CON|")
^$DEVICE("|NULL|")
```

MiniM does not support read value, assign value and *kill* command with ^$DEVICE variable.

9.2 $GLOBAL

Structured system variable ^$GLOBAL exposes information about globals with data.

Syntax

^$G[LOBAL](globalname)

Definition

globalname - global name. Structured system variable ^$GLOBAL supports only one subscript level.

First subscript of ^$GLOBAL must be a global variable name. It does not supported global variable name with subscripts, in this case process generate an error <SSVNSUBSCRIPT>. Global name can be specified with or without database name using extended syntax. If global name specified without database name, it is call to current database globals list, otherwise it is call to specified database globals list. If specified database does not exists, all calls generate an error <NAMESPACE>.

There is possible to use structured system variable ^$GLOBAL as an argument of functions *$data()*, *$order()*, *$query()* and as argument of command *kill*.

^$GLOBAL as an argument of *$data()* function

Function *$data()* with a structured system variable ^$GLOBAL as an argument return information about exist this global or not. For existing globals function returns value 1, otherwise 0. MiniM suppose global is existing global if at least root global name or one subscripted name have any data, otherwise global does not exists. For example:

```
TEMP>w $d(^$G("^a"))
0
TEMP>w $d(^$G("^|""aa""|a"))

<NAMESPACE>
TEMP>w $d(^$G("^a(123)"))

<SSVNSUBSCRIPT>
```

Structured system variable is used in most cases as globals directory list.

^$GLOBAL as an argument of functions *$order()* and *$query()*

Functions *$order()* and *$query()* allow to enumerate available globals. Functions can be used with specifying enumerate direction - forward or backward. If next global name does not found, functions returns an empty strings. To get first or last global name need to be used an empty sring as start value.

Function *$order()* returns next global name, and function *$query()* returns name of ^$GLOBAL with global name as first subscript.

If functions get start values with database name, functions search next global name in specified database, otherwise functions search in current database.

MiniM Database Server supports globals name mappint using the followint rules:

1) Globals with names starts with "%" symbol are physically stored in database "%SYS" and are accessible by all processes from any database.

2) Globals with names starts with "mtemp" string are physically stored in database "TEMP" and are accessible by all processes from any database.

On enumerating globals names using ^$GLOBAL it is used this conventions and functions *$order()* and *$query()* can return not only globals in current database.

If function *$query()* accept global name with database name specified, function returns next global with database name too, otherwise without database name. Database name is returned in upper case and embraces with || symbols even if source database was specified in lower case and was embraced with [] symbols.

If functions *$order()* and *$query()* got global name with subscripts or invalid global name, it generate an error <SSVNSUBSCRIPT>.

^$GLOBAL as a *kill* command argument

Structured system variable ^$GLOBAL can be used as *kill* command argument. In this case command remove entire global specified in ^$GLOBAL first subscript, for example:

```
USER>w ^a
a
USER>k ^$G("^a")

USER>w ^a

<UNDEFINED>
```

In this case *kill* with a ^$GLOBAL as an argument is equal of *kill* command with a first subscript of ^$GLOBAL as an argument.

```
kill ^a
```

9.3 $JOB

Structured system variable ^$JOB exposes information about available jobs in current MiniM instance.

Syntax

^$J[OB](jobnumber)

Definition

First subscript of ^$JOB is a *jobnumber*, a MiniM job identifier. Values are the same as returns system variables *$job*, *$zparent* and *$zchild*. Structured system variable ^$JOB supports only one subscripts level.

There is possible to use structured system variable ^$JOB as an argument of functions *$data()*, *$order()*, *$query()*, *$get()* and as argument of command *kill*.

^$JOB as a *$data()* function argument

Function *$data()* with a structured system variable ^$JOB as an argument return information about exist this job or not. For existing jobs function returns 1 otherwise 0.

```
TEMP>w $d(^$j($j))
1
TEMP>w $d(^$j("fdf"))
0
```

^$JOB as an argument of *$order()* function

Function *$order()* with a structured system variable ^$JOB as an argument allow to enumerate available jobs of this MiniM instance. Function can be used with specifying enumerate direction - forward or backward. If next job number does not exists, function return an empty string. To get job number with smallest or biggest number need to be used an empty string as start job number.

```
TEMP>s j="" f  s j=$o(^$j(j)) q:j=""  w j,!
1660
1250
```

^$JOB as an argument of *$query()* function

Function *$query()* with a ^$JOB as an argument allow to enumerate names of ^$JOB variable with job numbers as first subscript. Function *$query()* with a ^$JOB can be used with a direction specified - forward or backward.

```
TEMP>s n=$na(^$j("")) f  s n=$q(@n) q:n=""  w n,!
^$JOB("1608")
```

^$JOB as a *kill* command argument

MiniM Database Server allow to use a structured system variable ^$JOB as an argument of *kill* command. In this case first ^$JOB subscript must be a job number to terminate. It is not allowed to kill self, this generate an error <JOB>.

Process to be terminated got signal to terminate, execute *halt* command and exits.

Read data from ^$JOB variable

MiniM Database Server allow to read from ^$JOB variable. If first subscript is a job number of existing job, value of ^$JOB(pid) return 4 process internals as list in *$listbuild()* format. Order of elements:

1	Current database. ($znspace)
2	Current routine. ($zname)
3	Current device. ($principal)
4	Current naked indicator. ($zreference)

Some elements of this list can be undefined, and it is need to use *$listget()* function.

If evaluates value of ^$JOB(pid) for unexisting job, it generate an error <SSVN VALUE>.

If function *$get()* is used to evaluate value of ^$JOB(pid), function return specified default value or an empty string if this process does not exists.

9.4 $LOCK

Structured system variable ^$LOCK exposes information about available locks in current MiniM instance. Variable allow to enumerate available locks, return information about process owned and lock count and remove locking.

Syntax

^$L[OCK](lockname)

Definition

lockname - expression evaluated as a string and contain a local or global variable locked. Variable name can be with subscripts and global names with database name.

For global variable it is a rule for database name - if database name of global is omitted or it is an empty string, it equal current database usage and database names are case insensitive. Functions which can return global name including functions with ^$LOCK with an argument can return global name with database, and in this case database name returns in upper case and embraced into || symbols even if source database was specified in lower case and embraced into [] symbols.

There is possible to use structured system variable ^$LOCK as an argument of functions *$get()*, *$data()*, *$order()*, *$query()* and as a *kill* command argument.

^$L[OCK] as an argument of *$data()* function

Function *$data()* with a structured system variable ^$LOCK as an argument return information about exist this lock or not. For existing locks function returns 1 otherwise 0. Lock exist if this variable is locked by any process.

Function *$data()* with a structured system variable ^$LOCK as an argument return full indicator - is specified name is locked and is locked subscripted name or not.

```
TEMP>l

TEMP>w $d(^$l("^a"))
0
TEMP>l +^a

TEMP>w $d(^$l("^a"))
1
TEMP>l +^a(1)

TEMP>w $d(^$l("^a"))
11
TEMP>l -^a
```

```
TEMP>w $d(^$l("^a"))
10
```

^$L[OCK] as an argument of *$order()* function

Function *$order()* with a structured system variable ^$LOCK as an argument allow to enumerate currently locked by any process variable names in current MiniM instance. Variable names are returned as is, and global variables are returned using extended syntax, with database name specification. Local variable names are sorted prior by global names.

```
TEMP>l +a,+^a,+b,+^b

TEMP>s n="" f  s n=$o(^$l(n)) q:n=""  w n,!
a
b
^|"TEMP"|a
^|"TEMP"|b
```

Function *$order()* with a structured system variable ^$LOCK as an argument allow to use enumerate direction - forward or backward. And, if next locked name does not exists, function returns an empty string.

```
TEMP>s n="" f  s n=$o(^$l(n),-1) q:n=""  w n,!
^|"TEMP"|b
^|"TEMP"|a
b
a
```

^$L[OCK] as an argument of *$query()* function

Function *$query()* with a structured system variable ^$LOCK as an argument allow to enumerate all available ^$LOCK variable names with locked variable names as first subscript. Function *$query()* with a structured system variable ^$LOCK as an argument allow to use enumerate direction - forward or backward. And, if next locked name does not exists, function returns an empty string.

```
TEMP>s n=$na(^$l("")) f  s n=$q(@n) q:n=""  w n,!
^$LOCK("a")
^$LOCK("b")
```

```
^$LOCK("^|""TEMP""|a")
^$LOCK("^|""TEMP""|b")

TEMP>s n=$na(^$l("")) f  s n=$q(@n,-1) q:n=""  w n,!
^$LOCK("^|""TEMP""|b")
^$LOCK("^|""TEMP""|a")
^$LOCK("b")
^$LOCK("a")
```

^$L[OCK] as an argument of *$get()* function and evaluating value

MiniM Database Server allow to use ^$LOCK variable as an argument of *$get()* function and allow evaluating value of ^$LOCK variable with locked variable name as first subscript. This operation returns a process number owned by lock and lock count made delimited by colon.

```
TEMP>l +a w ^$LOCK("a")
1660:1
TEMP>l +a,+a

TEMP>w ^$LOCK("a")
1660:3
```

If specified lock name does not exist, function *$get()* return specified default value or an empty string.

```
TEMP>l

TEMP>w ^$LOCK("abcd")

<UNDEFINED>
TEMP>w $g(^$LOCK("abcd"),"undef")
undef
```

^$L[OCK] as a *kill* command argument

MiniM Database Server allow to use ^$LOCK variable as a *kill* command argument. In this case process removes lock on specified in first subscript variable. Lock is removed even id this lock is made by other process. For example:

```
k ^$LOCK("a")
```

9.5 $ROUTINE

Structured system variable ^$ROUTINE exposes information about available routines in the current database.

Syntax

^$R[OUTINE](routinename)

Definition

routinename - expression evaluated as a string and must be a routine name without circumflex (^). Structrured system variable ^$ROUTINE supports only one subscripts level.

^$ROUTINE as an argument of *$data()* function

Function *$data()* with a structured system variable ^$ROUTINE as an argument return information about exist this routine in the database or not. For existing routines function returns value 1, otherwise returns value 0. For example:

```
$d(^$r("alpha"))
```

tests is present routine *alpha* in the database or not.

^$ROUTINE as an argument of *$order()* and *$query()* functions

Functions *$order()* and *$query()* allow to enumerate routines available in the database. Functions can use enumerating directions - forward or backward. If next routine name does not exists, functions return an empty strings. To get first or last routine name need to be specified empty string as first subscript of ^$ROUTINE.

Functions enumerate only routines from current database, and does not use routines from the "%sys" database even if they are accessible to execute.

To enumerate system routines process must change current database to the "%sys" database.

^$ROUTINE as an argument of *kill* command

Command *kill* with a structured system variable ^$ROUTINE as an argument remove source code of routine specified in forst subscript. Routine's bytecode still exists and can be used to execute routine later. For example:

```
kill ^$r("alpha")
```

it remove source code of routine *alpha*.

Chapter 10

Device Parameters

10.1 COM

Devices with the |COM| type are supported only in the Windows editions of the MiniM Database Server and MiniMono.

The |COM| device used to interoperate with computer's serial COM ports, control port state and transfer data. To read from and write to port used *read* and *write* commands. To control state - *use* command with parameters and for special cases system function *$view("dev")*.

The |COM| device name need to be in the form "|COM|N", where N is a serial port number. For example, to interoperate with port COM2 need to be used device name

```
"|COM|2"
```

The port number must be specified withot leading zeroes.

The Windows operating system allow to processes operate with the same port concurrently with several exceptions, dependent of hardware. For example, modem as Internet provider or mouse.

With opening using *open* command or with *use* command can be used the following device parameters. With *close* command this parameters are ignored.

|COM| device parameters

Parameter /MODE=expr

The /MODE parameter assign device mode. The *expr* expression evaluates as string and used as a symbolic flags sequence. Used the following flags:

r or R	Enable read from device.
w or W	Enable write to device.
t or T	Enable text mode for read and write.
b or B	Enable binary mode for read and write.
other	Ignored.

By default device use mode as "rt", read is enabled, write is disabled and enabled text mode.

If text or binary mode does not specified, it is used text mode by default. If /MODE parameter is specified without read or write flags, both commands will generate <READ> and <WRITE> errors correspondingly.

When the text mode is enabled, then the line feed format

```
write !
```

outputs to device symbols $c(13,10), and if binary mode is enabled, it outputs symbol $c(10) only.

The /MODE parameter can be specified by position in first position or by name in any position.

Parameter /TERM=expr

The /TERM parameter defined the read terminator for binary mode. The *expr* expression evaluates as string and used as a terminator later. On read string terminator does not included into read result.

For text mode the read terminator is ignored and used special symbol $c(10). If before this symbol present symbol $c(13), it removes from result string too.

By default the terminator value is empty string.

The /TERM parameter can be specified by position in second position or by name in any position.

Other specified parameters can be used only in named mode, not by position.

Parameter /BAUD=expr

The expr evaluates as an integer. Specifies the baud rate at which the communications device operates. This value can be an actual baud rate value, or one of the following baud rate indexes: 110, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 56000, 57600, 115200, 128000, 256000. For values less then 110 and greater than 256000 generates <DEVPARAM> error.

Parameter /PARITYCHECK=expr

Specifies whether parity checking is enabled. If *expr* evaluates as nonzero, parity checking is performed and errors are reported.

Parameter /CTS=expr

Specifies whether the CTS (clear-to-send) signal is monitored for output flow control. If *expr* evaluates as nonzero and CTS is turned off, output is suspended until CTS is sent again.

Parameter /DSR=expr

Specifies whether the DSR (data-set-ready) signal is monitored for output flow control. If *expr* evaluates as nonzero and DSR is turned off, output is suspended until DSR is sent again.

Parameter /DSRS=expr

Specifies whether the communications driver is sensitive to the state of the DSR signal. If *expr* evaluates as nonzero, the driver ignores any bytes received, unless the DSR modem input line is high.

Parameter /TXC=expr

Specifies whether transmission stops when the input buffer is full and the driver has transmitted the XoffChar character. If *expr* evaluates as nonzero, transmission continues after the input buffer has come within XoffLim bytes of being full and the driver has transmitted the XoffChar character to stop receiving bytes. If *expr* evaluates as zero, transmission does not continue until the input buffer is within XonLim bytes of being empty and the driver has transmitted the XonChar character to resume reception.

Parameter /ABORTONERROR=expr

Specifies whether read and write operations are terminated if an error occurs. If *expr* evaluates as nonzero, the driver terminates all read and write operations with an error status if an error occurs. The driver will not accept any further communications operations until the error reason removed and *$zb* system variable called.

Parameter /OUTX=expr

Specifies whether XON/XOFF flow control is used during transmission. If *expr* evaluates as nonzero, transmission stops when the XoffChar character is received and starts again when the XonChar character is received.

Parameter /INX=expr

Specifies whether XON/XOFF flow control is used during reception. If *expr* evaluates as nonzero, the XoffChar character is sent when the input buffer comes within XoffLim bytes of being full, and the XonChar character is sent when the input buffer comes within XonLim bytes of being empty.

Parameter /NULL=expr

Specifies whether null bytes are discarded. If *expr* evaluates as nonzero, null bytes are discarded when received.

Parameter /DTR=expr

Specifies the DTR (data-terminal-ready) flow control. The *expr* value can be one of the following values:

0	Disables the DTR line when the device is opened and leaves it disabled.
1	Enables the DTR line when the device is opened and leaves it on.
2	Enables DTR handshaking.

On the other *expr* values process generates <DEVPARAM> error.

Parameter /RTS=expr

Specifies the RTS (request-to-send) flow control. The *expr* value can be one of the following values:

0	Disables the RTS line when the device is opened and leaves it disabled.
1	Enables the RTS line when the device is opened and leaves it on.
2	Enables RTS handshaking. The driver raises the RTS line when the "type-ahead" (input) buffer is less than one-half full and lowers the RTS line when the buffer is more than three-quarters full.
3	Specifies that the RTS line will be high if bytes are available for transmission. After all buffered bytes have been sent, the RTS line will be low.

By default it is used as /RTS=2. On the other *expr* values process generates <DEVPARAM> error.

Parameter /BYTESIZE=expr

Specifies the number of bits in the bytes transmitted and received. The

expr value can have one of the following values: 5, 6, 7, 8. On the other *expr* values process generates <DEVPARAM> error.

Parameter /PARITY=expr

Specifies the parity scheme to be used. The *expr* value can be one of the following values:

0	None
1	Odd
2	Even
3	Mark
4	Space

On the other *expr* values process generates <DEVPARAM> error.

Parameter /STOPBITS=expr

Specifies the number of stop bits to be used. The *expr* value can be one of the following values:

1.0	1 stop bit
1.5	1.5 stop bits
2.0	2 stop bits

On the other *expr* values process generates <DEVPARAM> error.

The use of 5 data bits with 2 stop bits is an invalid combination, as is 6, 7, or 8 data bits with 1.5 stop bits.

Parameter /XONLIM=expr

Specifies the minimum number of bytes allowed in the input buffer before the XON character is sent. Maximum value of *expr* is 64K.

Parameter /XOFFLIM=expr

Specifies the maximum number of bytes allowed in the input buffer before the XOFF character is sent. The maximum number of bytes allowed is calculated by subtracting this value from the size, in bytes, of the input buffer. Maximum value of *expr* is 64K.

Parameter /XONCHAR=expr

Specifies the value of the XON character for both transmission and reception. The *expr* value evaluates as a string and used only first symbol. If value is empty string, process generates a <DEVPARAM> error.

Parameter /XOFFCHAR=expr

Specifies the value of the XOFF character for both transmission and reception. The *expr* value evaluates as a string and used only first symbol. If value is empty string, process generates a <DEVPARAM> error.

Parameter /ERRORCHAR=expr

Specifies the value of the character used to replace bytes received with a parity error. The *expr* value evaluates as a string and used only first symbol. If value is empty string, process generates a <DEVPARAM> error.

Parameter /EOFCHAR=expr

Specifies the value of the character used to signal the end of data. The *expr* value evaluates as a string and used only first symbol. If value is empty string, it is used character $c(0).

Parameter /EVTCHAR=expr

Specifies the value of the character used to signal an event. The *expr* value evaluates as a string and used only first symbol. If value is empty string, process generates a <DEVPARAM> error.

System variable *$za* value

If current device is a |COM| device, system variable $za returns set of bit of port event's masks within one integer. Masks are single bits in separate positions of binary representation of integer. And common integer value is a sum of all available masks. To select special bit we can use arithmetic operators \ and #. For example, let it be sum of two numbers: #10+#20 (16+32). And, to determine separate bits we can use operators (here use hexadecimal numbers):

```
USER>w (#10+#20\#10#2)
1
USER>w (#10+#20\#20#2)
1
USER>w (#10+#20\#40#2)
0
USER>w (#10+#20\#8#2)
0
```

Here was determined that inside integer 48 are present masks #10 and #20, but are not presents masks #40 and #8.

Events mask list:

#0040	A break was detected on input.
#0008	The CTS (clear-to-send) signal changed state.
#0010	The DSR (data-set-ready) signal changed state.
#0080	A line-status error occurred, frame, overrun or parity error.
#0100	A ring indicator was detected.
#0020	The RLSD (receive-line-signal-detect) signal changed state.
#0001	A character was received and placed in the input buffer.
#0002	The event character (first character of EVTCHAR option) was received and placed in the input buffer.
#0004	The last character in the output buffer was sent.
#0800	An event of the first provider-specific type occured
#1000	An event of the second provider-specific type occured.
#0200	A printer error occured.
#0400	The receive buffer is 80 percent full.

System variable *$zb* value

If current device is a |COM| device, system variable $za returns set of bit of port error's. Error's mask list:

#0010	The hardware detected a break condition.
#0008	The hardware detected a framing error.
#8000	The requested mode is not supported. If this bit is present, other bits have not any meaning.
#0002	A character-buffer overrun has occurred. The next character is lost.
#0001	An input buffer overflow has occurred. There is either no room in the input buffer, or a character was received after the end-of-file (EOF) character.
#0004	The hardware detected a parity error.
#0100	The application tried to transmit a character, but the output buffer was full.

To determine what bit is present, we can use the same methods as described in $za value unpacking. Note each call to $zb variable for |COM| port make error mask read and clearing. Other call to $zb can return other value.

Additional COM port control

Additionally to control COM port state MiniM implements several special extended functions $v("dev",5), $v("dev",6) and $v("dev",7). See function description in chapter about $view function.

The *write #* command

If the *write #* command applied to COM port, it is flushes input and output buffers.

The *write ?N* command

If the *write ?N* command applied to COM device, it writes N spaces. If specified less then 1 ot more the 32K spaces it does nothing.

For example, how to read modem identification strings if modem is connected to COM2 port:

```
USER>k  s dev="|COM|2" o dev

USER>u dev:(/BYTESIZE=8:/BAUD=115200:/MODE="rwt") >>
  >> w "ATI3",$c(13,10),# r ans,ans,#
USER>c dev

USER>w
ans="U.S. Robotics 56K FAX EXT Rev. 11.15.19"
dev="|COM|2"
```

Here read and write operations are terminated by flush buffers action. Here set up one of standard speed, byte size as 8 bits and text mode to interact. Next the read commands read acho and identification string.

10.2 CON

The CON device is created automatically on process start in console mode. This mode is used by default if it is not telnet-process, jobbed process, no any redirection of input and output is used and no specified STD option. For example, if we run MiniM process directly from cmd utility or from Windows Explorer:

```
minim.exe
```

in this case MiniM process use CON device as principal and current:

```
W:\MiniM\bin>minim.exe
TEMP>w $p
|CON|
TEMP>w $io
|CON|
TEMP>
```

This device cannot be created manually except automatically and principal device cannot be changed to another. MiniM process in console mode use Windows console functionality to handle input and output.

The CON device have not *close* command parameters (closing principal device always ignored) and have no any *open* parameters (this device cannot be opened directly).

Use command options / parameters

The CON device with *use* command support the following options:

/COLUMNS = expr	Use specified columns for console window.
/LINES = expr	Use specified lines for console window.
/ECHO = expr	Use or not use echo mode for read command.
/TERM = expr	Read terminator
/OEM = expr	Use automatic OEM - ANSI decoding
/CTRLC = expr	Trap or not on press Ctrl+C
other	Ignored.

Option /COLUMNS = expr

The /COLUMNS = expr option specifies how many columns need to be used for console window. By default is used 80 columns.

The /COLUMNS = expr option can be specified by position in first position on by name in any position.

The *expr* value evaluates as integer, if value is less then 1, process generates the <DEVPARAM> error.

Option /LINES = expr

The /LINES = expr option specifies how many lines need to be used for console window. By default is used 25 lines.

The /LINES = expr option can be specified by position in 2 position or by name in any position.

The *expr* value evaluates as integer, if value is less then 1, process generates the <DEVPARAM> error.

Option /ECHO = expr

The /ECHO = expr option specifies echo mode usage. By default echo is used - read command display on the screen all entered symbols.

The *expr* value evaluates as an integer. If it is 0, echo mode is turned off, otherwise on. If echo is disabled, entered symbols does not shown on the console screen.

The /ECHO option can be specified by position in 3 position or by name in any name.

Option /OEM = expr

The /OEM = expr option control OEM to ANSI encoding usage. By default OEM encoding is enabled and device uses OEM codepage to represent data in the console window (Windows behavior by default) and internal data in ANSI encoding (Windows collation by default). The *expr* value evaluates as an integer. If is it zero, encoding is disabled, otherwise enabled. The /OEM option is supported by name only.

Examples:

```
use $p:/OEM=0   ; disable OEM -> ANSI encoding
use $p:/OEM=1   ; enable OEM -> ANSI encoding
```

Option /CTRLC = expr

The /CTRLC = expr option enable or disable <INTERRUPT> error generation on Ctrl+C press. If *expr* evaluates as zero, error generation is disabled and Ctrl+C press can be read from keyboard using *read* command. If it is non zero, Ctrl+C press generates <INTERRUPT> error. The /CTRLC = expr option can be used only by name.

Option /TERM = expr

The /TERM = expr option specifies read terminator. Read string terminates if this character sequence is accepted. The terminator value does not included into read string result. The /TERM option for console can be specified by name only.

If expr evaluates as an empty string, terminator does not used.

In interactive mode while command enters the terminator does not forget and used later in read string command. Top command input mode use only carrage return as input terminator.

Read terminators

In console read string terminator by default is carrage return symbol (ENTER) and escape symbol (Esc). This terminators used if /TERM option spesifies empty string. On the non-zero stack execution id terminator is specified, it is used, otherwise used ENTER and Esc as symbols to end of read string.

On top level (zero stack level) it is work internal string enter editor with scrolling and it is form current string to read.

On read character command console device use direct console buffer read to accept symbols.

Additions to the $KEY value

On the read character command (read *char) MiniM adds to the $key value series of character to indicate that special keys (Ctrl, Alt or Shift) are pressed: "a", "c" and "s". If was pressed 2 special keys. added 2 symbols, and for all 3 keys added 3 symbols. Left and right Alt and left and right Ctrl does not destinquished. For example, on enter Ctrl+F we got:

```
USER>r *x s k=$k zzdump x,k
0000: 32 37                    27
0000: 1B 4F 50 63              .OPc
```

and on enter Ctrl+Shift+F1 we got:

```
USER>r *x s k=$k zzdump x,k
0000: 32 37                    27
0000: 1B 4F 50 63 73           .OPcs
```

As the functional keys are also used keys Insert, Delete, Home, End, Page Up, Page Down and arrow keys.

The console device return escape sequences for functional keys in the upper case and additional alternating modifiers in the lower case.

MiniMono difference

MiniM Embedded Edition does not implement —CON— device.

10.3 DLL

The —DLL— device is created automatically as a principal input-output device for MiniM Embedded Edition.

On the MiniMono initialization stage host process defines event handlers for this device. In the case of some handler was defined, the MUMPS routines use this handlers, otherwise MUMPS routines use default device behavior.

All device options and device behavior are fully defined by the host process.

10.4 FILE

The FILE devices used to create, delete, read and write files of OS file systems.

The FILE devices are identified by strings with the "|FILE|" prefix and following file name. The file devices are distinquished by file names and are case sensitive. This allows to use two devices to operate with one file, for example:

|FILE|c:/temp/dat.txt	File c:/temp/dat.txt
|FILE|\\srv/temp/dat.txt	File temp/dat.txt on server srv

The *open* command with FILE device ignores timeout value and cannot break file opening before timeout expired. But, if timeout is specified, process sets the $test value to 0 or 1 value if file does not opened or opened successfully.

MiniM supports for file devices several options by name and by position. If option is specified without name, it is used by position, otherwise by name in any position.

The *open* command options

/MODE	File open or usage mode.
/SHARE	Share mode.
/TERM	Read terminator.
/TRUNCATE	File truncation option.

The /MODE option

The /MODE=expr option requires the value of *expr*, and it is evaluates as a string and used as several character flags set. Characteds can be specified in any sequence and case. The /MODE option can be specified by position in first position.

r	Use file to read. Flag can be combined with the w flag.
w	Use file to write. Flag can be combined with the r flag.
a	Open file in append mode. On opening the file position automatically sets to the end of file.
t	Use file in text mode. Cannot be used with the b flag.
b	Use file in binary mode. Cannot be used with the t flag.
n	Open new only file.
e	Open existing only file.

When you open the file in binary mode terminator read default is empty string. If the read terminator is specified, it is used. If file is opened in text mode, the read terminator is $C(10) byte. If this byte is read, and last read byte is $C(13), this bytes are removed from read result. The $C(13) byte in other positions does not removed and read result does not incluse the read terminator. If file is opened in text mode, the /TERM option is ignored even if specified.

If both n and e flags does not specified, and w or a flags not specified, by default process openes only existing file without new file creation (as it was specified e flag). If was not specified n and e flags and flag w or a is present, process openes existing file or creates new file if this file does not exists.

Examples:

Open file to write and in append mode:

```
open "|FILE|c:\temp\text":("wa")
open "|FILE|c:\temp\text":(/MODE="wa")
```

Open existing file to read:

```
open "|FILE|c:\temp\text":("re")
open "|FILE|c:\temp\text":(/MODE="re")
```

If the /MODE option is not specified, by default used flags rte.

In text mode the carrage return command

```
write !
```

writes to the file two symbols $C(13,10). In binary mode this command writes only $C(10) symbol.

The /SHARE option

This option specifies file sharing mode with other processes. This option can be specified by position in 2 position or by name in any position. File sharing mode evaluates as a string and used as a share flags case insensitive.

r	Allow other processes read file.
w	Allow other processes write to file.
e	Deny other processes all access.

If this option is not specified, by default used file sharing flag r.

Examples:

Open file to read and allow other processes to read:

```
open "|FILE|c:\temp\dat":(:"r")
open "|FILE|c:\temp\dat":(/SHARE="r")
```

The /TERM=expr option

This option specifies the read string terminator for binary mode. If option is not specified, it is an empty string and read string command reads up to specified read length is reached or up to maximum string length (32K).

The /TERM=expr option can be specified by name in any position, or by position in 3 position for *open* command or in 2 position for *use* command.

For text mode the /TERM option is ignored.

The /TRUNCATE option

The /TRUNCATE option need to be specified for open command if file need to be truncated to zero file size. This option can be used only by name. For example:

```
open "|FILE|c:\temp\report23":("w":"e":/TRUNCATE)
```

Here the c:\temp\report23 file is opened for write and truncated to zero length.

The /TRUNCATE option requires that the file is opened for write or for append.

The *use* command options

The *use* command with the file device accept the following options:

/LOCK = expr	Make file locking. Used with the /SIZE option and optional /POS option.
/POS = expr	Specify file locking start position. Optional.
/SIZE = expr	Specify file locking size.
/OFFSET = expr	Specify file locking offset
/TRUNCATE	Truncate file to current position.
/TRUNCATE = expr	Truncate file to specified position.
/TERM = expr	Read string terminator.

The /LOCK = expr option

This option make file locking and expression *expr* evaluates as a string and must contain locking flags as character:

s or S	Make shared locking.
e or E	Make exclusive locking.
u or U	Unlock.
other	Generates <DEVPARAM> error.

Shared locking can be made by several processes and it deny to make exclusive locking. Exclusive locking can be made by one process and deny all other shared and exclusive locking.

On unlocking need to have start and size of locking area the same was locked.

One process can make several file locks on different file areas and can make different locking types. If file cannot lock specified area, process goes to wait locking until locking can be possible.

The /LOCK option can be specified only by name.

The /POS = expr option

The /POS = expr option is used in conjuction of the /LOCK option to setup start of locking region. This option can be used in conjuction with the /OFFSET option to specify /POS start countdoun. If the /POS option is not specified, process make lock from current file position.

The /POS option can be specified only by name.

The /SIZE = expr option

The /SIZE = expr option is used to specify file locking area size in conjuction with the /LOCK option.

The /SIZE option can be specified only by name.

The /OFFSET = expr option

The /OFFSET = expr option is used to specife the start point offset for the /POS option. The *expr* is evaluated as an integer and can be one of the following:

0	Start from file begin.
1	Start from current file position.
2	Start from file end.

Other option value generates the <DEVPARAM> error.

The /OFFSET option can be specified only by name.

Double meaning of /POS and /OFFSET options

If the /LOCK option is specified, a /OFFSET and /SIZE options applies to the file locking. If the /LOCK option is not specified, a /POS and /OFFSET options changes the current file position and the /SIZE option is ignored.

The /TRUNCATE option

The /TRUNCATE option without value make file truncation by current file position.

The /TRUNCATE option can be specified only by name.

The /TRUNCATE = expr option

The /TRUNCATE = expr option is used to specify file truncation by specified in the *expr* file position.

The /TRUNCATE = expr option can be specified only by name.

The /TERM = expr option

The /TERM = expr option is used to specify read string terminator. The value of the *expr* expression is evaluated as a string and used as a terminator. This terminator used only in the binary mode. For text mode terminator is saved but does not used.

The /TERM = expr option can be specified only by name.

The *close* command options

The *close* command with the file device accept the following options:

/TRUNCATE	Truncate file by current file position.
/TRUNCATE = expr	Truncate file by specified file position.
/DELETE	Delete file after closing.
/RENAME = expr	Rename file after closing.

The /TRUNCATE option

The /TRUNCATE option without value make file truncation by current file position.

The /TRUNCATE option can be specified only by name.

The /TRUNCATE = expr option

The /TRUNCATE = expr option is used to specify file truncation by specified in the *expr* file position.

The /TRUNCATE = expr option can be specified only by name.

Examples:

```
s dev="|FILE|c:\temp\log"
...
close dev:(/TRUNCATE)
```

The /DELETE option

This option deletes file after closing device.

The /DELETE option can be specified by name only.

The /RENAME = expr option

The /RENAME = expr option renames file after device closing. The value of *expr* is evaluated as a string and used as new file name.

The /RENAME option can be specified by name only.

Examples:

```
s dev="|FILE|c:\temp\log"
...
close dev:(/RENAME="c:\temp\log"_$h)
```

Since version 1.28 MiniM supports counting of system variables $X and $Y for FILE device in text mode (value "t" in option MODE).

10.5 MEM

The device with MEM type in interprocess communication device and operates by internal in-memory buffers. MiniM processes can exchange data using MEM devices and this data does not stored on the disk, does not

leave server and does not outputs to any other place. The MEM devices are bidirectional devices for two processes.

The MEM devices can have any name, followed by the device type prefix. Both processes to interact with device need to specify the same device names. First process who opens device first is a server process for this device. Second process who opens device is a client process for this device.

For MEM devices only one process can be a server process. If this server process closes device, this device is destroyed. Client process for the MEM device can be only one at one moment. Other client process can open device only after previous client process closes this device. If client process closes the MEM device, all data wrote to the memory buffer are lost if the server process does not read them.

If the client process does not open MEM device, the server process cannot write any data to device and write cimmands are wait until client process is connected.

If one client process closes MEM device, other process can open the same MEM device and be a current client process for this device. But this second client process cannot read data sent to first client process.

The MEM devices ignores *open* command timeout, but if timeout is specified, process sets the $test system variable to 0 or 1 in depend of device is opened and connected to or not successfully.

The *open* command options

The /MODE = expr option

This option specify the text or binary and read / write device mode. The expression *expr* evaluates as a string and need to be a set of special character flags. Characters are cese-insensitive.

This option can be used by position in 1 position or by name in any position.

Characters meaning:

r	Open device for read. Flag can be combined with the w flag.
w	Open device for write. Flag can be combined with the r flag.
t	Open device in text mode. Cannot be combined with the b flag.
b	Open device in binary mode. Flag cannot be combined with the t flag.

If device is opened in binary mode, the read string terminator by default is an ampty string. If terminator is specified, it is used. If edvice is opened in text mode, the read string terminator is $C(10) symbol. If before this symbol was read the $C(13) symbol, it is removed from string too. Symbols $C(13) in other positions does not removed. The read string command return string without terminator. In text mode the /TERM option is ignored even if it is specified.

The /MODE option always must have expression *expr*.

Examples:

```
s dev="|MEM|mnm srv sample"
o dev:("rwt")
o dev(/MODE="RWT")
```

If the /MODE opion is not specified, the MEM device by default use read and write permissions and work in binary mode.

The /MODE option can be specified differently by server and client process, each process operate by device with options which this process use.

The /TERM = expr option

The /TERM = expr option specify the read string terminator for binary mode. If this option is not specified, process use the empty string by default and read string command read data up to maximum string (32K) or up to specified read length.

This option can be specified by position in 2 position ot by name in any position.

The /TERM option always must have the *expr* expression.

For text mode the read string terminator does not used.

Server and client processes for MEM device can use different read string terminators.

The /BUF = expr option

This option specify internal data buffer for MEM device in bytes. If buffer is fullfilled by write process and other process does not read data, write process can stay in wait state to write data until buffer is read and have enough space.

Minimum buffer size is 512 and maximum buffer size is 1048576 bytes. If specified less then minimum, device use minimum size and if specified

greater than maximum, device use maximum. By default device use 65536 bytes buffer length.

Buffer size increasing grows the need server memory usage.

Buffer size cannot be changed by client process, and can be setting up only by creator process (server process).

The *use* command options

The *use* command for MEM devices can accept the following command options:

/TERM	Changes current read string terminator.
/MODE	Changes current read / write or text / binary mode.

Other *use* command options for MEM device are ignored.

The /MODE option for *use* command can be specified by position in 1 position and the /TERM option can be specified by position in 2 position.

The *close* command options

The *close* command for MEM devices does not accept any options. If any option have been specified, it is ignored. After closing device by client process the MEM device can accept connection from other process as client process. After closing device by server process (creator) the client process generate the <DEVICE> error.

MiniMono difference

MiniM Embedded Edition does not implement —MEM— device.

10.6 NULL

The NULL - type devices are devices with no any real input and output actions. Writing to the NULL device make data lost and read from NULL device return control immediately with no data. If process read string, command return empty string, if process read character code, command always return the -1 value.

Each process can open several NULL devices with different names followed by the device type prefix. For example:

```
TEMP>s dev="|NULL|" o dev

TEMP>s dev2="|NULL|2" o dev2
```

Here we open 2 NULL devices, with names "|NULL|" and "|NULL|2".

NULL devices are used to lost all output.

Open, use and close commands options

MiniM ignores all *open*, *use* and *close* commands options for NULL devices.

If new MiniM process is run by the *job* command, this child process by default have the NULL device as a principal and current device.

10.7 PIPE

The PIPE input-output devices runs console application with input-output redirection. Child process accept in stdin stream all *write* commands from MiniM process and MiniM process read data from child's stdout stream using *read* command.

Device identification string is a type prefix as "|PIPE|" with following child process file name with command line parameters. For example, to read file's list in the current folder can be used command line:

```
"|PIPE|dir /b"
```

Several Windows operating system utilities and other applications supports noninteractive execution mode and can exchange data with parent process using standard streams (stdin + stdout). For example, operating system command "dir" execution with different command line options can list directories to standatd output stream. If it is run in console mode, list displayed on the screen, and this output can be redirected to the file. The same possibilities used in the PIPE devices, but MiniM process interact with child process using *read* and *write* commands. For example:

```
>s dev="|PIPE|set" o dev
>u dev f  r a s s($i(s))=a
>c dev
```

Here we create PIPE device to execute *set* operating system command without command line options. All string was read are stored in lical variable *s*. Read terminates on error <ENDOFFILE>. This code mean that MiniM server settings TrapOnEof have value 1.

```
>s mode=$v("proc",5,0)
>s dev="|PIPE|set" o dev
>u dev f  r a q:$zeof  s s($i(s))=a
>c dev s mode=$v("proc",5,mode)
```

This example directly control TrapOnEof mode to read all strings from child process and code control the $zeof system variable value to check all data was read. After full reading process returns previous TrapOnEof state.

The *open* command options

With PIPE devices the *open* command supports followinf options:

/MODE = expr	Open mode - text / binary, read / write.
/TERM = expr	Read string terminator for binary mode.
/DIR = expr	Child process current directory.

The /MODE = expr option

This option specify the device mode after opening. The *expr* expression evaluates as a string and need to be a set of special flags case insensitive. Option supports following flags:

r or R	Enable read from device. Flag can be combined with w flag.
w or W	Enable write to device. Flag can be combined with r flag.
t or T	Use text mode to read and write. Flag cannot be combined with b flag.
b or B	Use binary mode to read and write. Flag cannot be combined with t flag.
other	Ignored.

If /MODE option is not specified, device openes in rt mode, Read enabled, write disabled, text mode for *read* and *write* commands.

If no r or b flag specified, by default used text mode. If does not specified r or w flags, all read and write commands are disabled and generates <READ>

and <WRITE> errors correspondingly.

If text mode is enabled, line feed write

```
write !
```

outputs to device symbols $c(13,10), and in binary mode only $c(10) symbol.

The /MODE option can be specified by position in 1 position and by name in any position.

The /TERM = expr option

This option specifies read string terminator for binary mode. Expression *expr* evaluates as a string. After read string terminator is removed from read result.

For text mode the read string terminator is ignored. The text mode read string terminator is a $c(10) symbol. If before this symbol was read $c(13) symbol, it removes from read result too.

By default read string terminator is an empty string.

The /TERM option can be specified by position in 2 position or by name in any position.

The /DIR = expr option

This option specify the current directory for child process. The *expr* expression evaluates as a string and need to be existing directory name.

The /DIR option can be specified by position in 3 position or by name in any position.

By default child process runs in the current MiniM process directory.

The *use* command options

The *use* command with PIPE device accepts only one option - terminator change (/TERM = expr). This option changes previous read string terminator and use if current mode is bunary mode.

The /TERM option for *use* command with PIPE device can be used only by name in any position.

The *use* command cannot change read / write or text / binary mode.

Other options are ignored.

The *close* command options

The *close* command for PIPE device accepts only one option, /TERMINATE without value. If /TERMINATE option is present, MiniM process terminates child process if it is active.

The /TERMINATE option can be specified only by name in any position.

Other options are ignored.

10.8 PRN

Devices with the |PRN| type are supported only in the Windows editions of the MiniM Database Server and MiniMono.

The PRN device is used for printing and to control printing options.

This devices are work only in write mode, all *read* commands generates a <READ> error.

Print control options for PRN devices are accepted only with *open* command. The *use* and *close* commands cannot change printing options.

Some printing devices which works as a printer but are not read hardware printers (for example, virtual printers or faxes) can create dialog windows to detail some printing options, for example to specify pdf file name. Programmer must check every used printer type for used user account and check printer driver behavior. And programmer must check account's rights to print to network printers.

PRN device names

To specify printer need to be used a device identification string with "|PRN|" device type prefix with following printer name as it is specified in Windows. If it is an empty string

```
s printer="|PRN|"
```

MiniM process uses default printer. Printer names can be for example:

```
s dev="|PRN|Acrobat Distiller"
s dev="|PRN|Fax"
s dev="|PRN|EPSON TX117_119"
```

To specify network printer name need to be used the computer name and printer name as it is configured in the network, in UNC naming conventions. For example, if server name is "NTServer" and printer configured with "HP LaserJet 5P" network name:

```
s dev="\\NTServer\HP LaserJet 5P"
```

The *open* command options

All PRN device options can be specified only by name in any position and options by position does not supported.

The *open* command with PRN devices supports the following options (with depending of real printer driver possibilities):

/MODE	Setup text or binary mode.
/OUTPUTFILE	Specify the file name to output print image to.
/DOCNAME	Specify printer job name.
/ORIENTATION	Spacify printer paper oriantation.
/COPIES	Specify print copies number.
/QUALITY	Specify printing quality.
/COLOR	Specify coloring mode - color or black and white.
/DUPLEX	Specify duplex printing.
/COLLATE	Specify printed pages collation method.
other	Ignored.

The /MODE = expr option

The value of *expr* expression evaluates as a string. If this string contains a symbol "t" or "T", device opens in a text mode. Id string contains a symbol "b" or "B", device opens in a binary mode. Binary mode corresponds to the "RAW" printer format and text mode corresponds to the "TEXT" printer format. If this option does not specified, or *expr* does not contain special mode flags, process opens device in a text mode, it is by default.

The /OUTPUTFILE = expr option

If this option is specified, device outputs to file name specified in *expr* value. If this option is not specified, printer driver use default settings.

The /DOCNAME = expr option

If this option is specified, the *expr* expression value used as a printer job name. If this option is not present, MiniM process use default printer job name "MiniM".

The /ORIENTATION = expr option

If this option is specified, the *expr* expression evaluates as a string. MiniM supports the following values:

"PORTRAIT"	Print with vertical page orientation.
"LANDSCAPE"	Print with horizontal page orientation.
other	Ignored and printer use default settings.

The *expr* value used case insensitive.

The /COPIES = expr option

If this option is not specified, printer driver uses printing copies number configured by default. If option is present, *expr* expression evaluates as an integer an need to be greater then 0. Otherwise MiniM process use the value of 1. This option specify printing copies number to print in this job. This option usage depends of printer driver possibilities.

The /QUALITY = expr option

This option specify printing quality. If option is not present, printer driver uses the value configured by default. If present, the value of *expr* expression need to be one of the following strings: "HIGH", "MEDIUM", "LOW", "DRAFT". This values used case insensitive. Other values are ignored and printer driver uses the default settings. This option usage depends of printer driver possibilities.

The /COLOR = expr option

This option specify coloring option. If option does not specified, printer driver uses default settings. If option is present, the *expr* expression evaluates as a string and can be one of the following (case insensitive):

"COLOR"	Print using colors.
"MONOCHROME"	Print using black and white or in grayscale mode if possible.
other	Ignored and printer driver uses default settings.

This option usage depends of printer driver possibilities.

The /DUPLEX = expr option

This option specifi printing using one or both page sides (if hardware allow this). If used landscape page orientation, both top page edges are on the same page edge. If used portrait pahe orientation, bottom page edge is on the same page of top edge of next page. The value of option can be:

"Simplex"	One-side printing with current page orientation.

"Horizontal"	Two-side printing with landscape page orientation.
"Vertical"	Two-side printing with portrait page orintation.
other	Ignored and printer driver uses default settings.

This option values are used case insensitive.

The /COLLATE = expr option

This option specify printed copies collation (if hardware supports this). The value of *expr* evaluates as an integer and compares with 0. If it is 0, copies collation is disabled for this printer job, otherwise enabled. If this option does not present, printer driver uses default settings.

The *write* command specific behavior for the PRN device

write !

In text mode this command outputs symbols $c(13,10), and in binary mode only $c(10) symbol.

write #

This command is equivalent of output page feed command ($C(12)). Programmers should consider this in binary mode.

write ?N

In text mode this command outputs appropriate space numbers to feet to specified position *N*. In binary mode this command outputs *N* spaces.

The *close* command for PRN device make printer job ending commands and ends printing.

10.9 STD

The STD device is creates automatically if MiniM process runs with input-output redirection or with -std option.

If process runs with redirection, process get commands from standard input and terminates after execution last command.

If MiniM process runs with -std command line option, process check option -x is present and executes specified commands. After commands execution process check option -h present. If -h command line option is present,

process terminates execution, otherwise still read next commands from standard input stream.

The STD device cannot be created manually and cannnot be closed because it is principal device. The STD device can be controlled by only *use* commands only.

The *use* command option

The *use* command option with STD device can accept the following options:

/TERM = expr	Specify read string terminator.
/MODE = expr	Specify input-output mode - text or binary.
other	Ignored.

The /MODE = expr option

The /MODE option specify input-output mode. The *expr* expression evaluates as a string and value can contain one character of the following:

t or T	Switch device to text mode.
b or B	Switch device to binary mode.
other	Generates a <DEVPARAM> error.

In text mode read string terminator is a symbol $C(10). If before this symbol was read a symbol $C(13). this symbol removes from read result too.

By default STD device still work in a text mode.

While proicess read commands from input the device works in a text mode and before execution commands switches to specified mode.

In binary mode device use specified terminator. By default device have empty string as a terminator.

The STD device does not support read timeouts and specified in *read* commands timeouts are ignored. But if a timeout is specified, the system variable $test sets into value 1.

MiniMono difference

MiniM Embedded Edition does not implement —STD— device.

10.10 STORE

The STORE device uses data in local or global variables. All write commands appends data to variable and read commands read data from variable. Local or global variables of STORE device are automatically indexed if need to store long data or many strings.

The STORE device identification string is a "|STORE|" prefix with following local or global variable name. This variable can have one or more indises. For example:

```
s dev1="|STORE|stream(""""")"
s dev2="|STORE|"_$na(^mtempRun($j,""))
```

The STORE device automatically switches between indices using internal $order function call. Start value of last used index is a start value for this $order step. This step is automatically made on device open. On *read* commands device uses indices as is, what exists in variable and *write* command automatically increment next index by one.

The STORE device can be opened for read only or for write only. Before switch to write mode device removes all data from specified local or global variable after specified last index.

All *read* and *write* commands are made in current transaction context and internally use the *set* commands with all need side effects.

The STORE commands options

The /MODE=expr option

The /MODE option specify device mode. The *expr* expression evaluates as a string and is used as a special symbol flags. Device supports the following flags:

r or R	Open device to read. Cannot be combined with w flag.
w or W	Open device to write. Cannot be combined with r flag.
t or T	Switch to text mode. Cannot be combined with b flag.
b or B	Switch to binary mode. Cannot be combined with t flag.
other	Ignored

If this option does not specified, device use by default flags "wt", allow write and still work in text mode and before opening device automatically removes all data after specified device name.

The /MODE option can be used by position in 1 position or by name in any position for *open* and *use* commands.

In the binary mode the command

```
write !
```

writes to variable symbol $c(10) and in text mode writes symbols $c(13,10).

The /TERM = expr option

This option specify read string terminator for binary mode. The *expr* expression evaluates as a string and used as read terminator. The terminator content does not included into read result.

For text mode the read string terminator is ignored. For text mode as read string terminator uses symbol $c(10). If before this symbol was read symbol $c(13), this symbol removes from read result too.

The read string terminator by default is empty string.

The /TERM option can be specified by position in 2 position or by name in any position.

The /EOL = expr option

The /EOL option specify how to handle end-of-line command in text mode. This option are applied only in text mode. The *expr* expression evaluates as an integer and compares with 0. If value is not zero, in text mode all read commands use end of current subindex as end of string and *write !* command automatically switches to next subindex. Otherwize *read* and *write* commands use subindices values sequentially as one very big character sequence.

By default this option is 1.

This option can be specified by position in 3 position or by name in any position for *open* and *use* commands.

The *close* command with STORE device ignores all options.

Example how to use the STORE device:

```
USER>s dev="|STORE|abc(12)" o dev

USER>u dev w 456,!,789 u 0 c dev w
abc(13)="456"
abc(14)="789"
dev="|STORE|abc(12)"
```

Here we open STORE device to write, in text mode and use end-of-line as new subindex ((/MODE="wt":/EOL=1)). And here we specify to use local variable *abc* with subindex *12*. Next we make this device current and write string 456, end-of-line and string 789. Next we close this device. After this actions we have written data into local variables *abc(13)* and *abc(14)*.

10.11 TCP

The TCP device uses TCP/IP sockets to exchange data and control socket state. Device use open, receive, send data and close socket functions. MiniM device can open TCP device as client side as such as server side socket.

TCP device have device identification string as "|TCP|" device type with following tcp address and port or computer name and port or port only. MiniM process uses full device name to determine is it a server-side socket or a client-side socket. Device name are distinquished by name cese sensitive and programmer must check case of computer name to prevent or allow usage connection to the same tcp server.

Naming conventions

—TCP—	Device has been created automatically for jobbed MiniM process with *job* command and with concurrent socket.
—TCP—:port	Device is a server-side socket for this port.
—TCP—server:port	Device is a client-side socket to connect to this server and port.
—TCP—address:port	Device is a client socket to connect to this address and port.

Examples:

```
s dev="|TCP|:80" o dev
```

```
s dev="|TCP|www.server.com:80" o dev
s dev="|TCP|127.0.0.1:80" o dev
```

Here we open 3 TCP devices. In 1) case we open server socket for port 80 and after opening device is ready to listen and accept incoming tcp connections. In 2) case we open client-side tcp socket to connect to server "www.server.com" and port 80. After opening device is ready to read and write data. In 3) case we open client device to connect to tcp address 127.0.0.1 and port 80. After opening device is ready to read and write data.

TCP device ignores *open* command timeout. But if timeout is specified, process change value of $test system variable to 0 if device does not opened or to 1 if all ok.

The *open* command options

/SOCKET	Specify tcp socket number already opened by other function inside current process.
/MODE	Specify text or binary and read - write mode.
/TERM	Specify read string terminator.

The /MODE = expr option

This option specify the device mode. The *expr* expression evaluates as a string and need to be special symbol flags case insensitive.

r	Open socket to read. Can be combined with w flag.
w	Open socket to write. Can be combined with r flag.
t	Open in text mode. Cannot be combined with b flag.
b	Open in binary mode. Cannot be combined with t flag.

The /MODE = expr option can be specified by position in 1 position or by name in any position.

Examples:

```
s dev="|TCP|:80"
o dev:("RW")
o dev(/MODE="RWT")
```

Here in 1) case we open TCP device to read and write in binary mode and option is specified by position. In the 2) case we open TCP device for read and write in text mode and option is specified bu name.

If the /MODE = expr option does not specified, TCP device use read and write permissions and still work in binary mode.

The /SOCKET = expr option

Syntax:

```
/SOCKET=socknumber
```

The /SOCKET option can be used to create TCP device with tcp socket has been opened by external functions inside this or external process, for example using $zdll function or if process runs from other process with socket inheriting. If new TCP device has been created with this option, device does not open new internal connection and use specified socket. On TCP device closing this socket closed too.

The /SOCKET option have not positional form usage and can be specified only by name. The *socknumber* expression always must be specified and evaluates as an integer.

The /TERM = expr option

This option specifies read string terminator for device in dinary mode. If this option has not been specified, device use empty string as read string terminator for binary mode and read up to maximum MiniM string length (32K) or to specified read string length limit.

In text mode device use as read string terminator the symbol $c(10). If before this symbol was read symbol $c(13), this symbol removes too from read string result. Other $c(13) symbols does not removed. Read string result returned without terminator. For text mode device accept, but does not use the /TERM option.

The /TERM option can be specified by position in 2 position ot by name in any position.

The value of *expr* expression evaluates as a string.

The *use* command option

The *use* command with TCP device accept the following options:

/TERM	Change current read string terminator.
/MODE	Change current device mode.
/ACCEPT	Goes to wait external connection.
/ATO	Accept timeout.

Other options the *use* command with TCP device ignores.

The /MODE option for *use* command can be used by position in 1 position and /TERM option van be used by position in 2 position. Other options can be used by name only.

The /ACCEPT option

The /ACCEPT option has not value. The *use* command for TCP device make going device to wait incoming tcp connection. On connection is made new socket stored in internal device data. If before this operation been accepted other incoming connection it is closed and replaced by new connection.

Accepted socket number can be obtained by $view("dev",2) function for current TCP device or by $view("dev",2,devname) for specified TCP device.

Acceptes TCP connection can be used by the *job* command to run one more MiniM process with concurrent socket on the same TCP port.

The /ATO = expr option

The /ATO option specify wait timeout to wait incoming TCP connection. Device mean that it was opened as a server socket. The /ATO option need to be with /ACCEPT option in pair and have mandatory value. This *expr* expression evaluates as a number and used a wait timeout in seconds.

This timeout is applied only to /ACCEPT operation and has a side effect. If incoming connection has abeen made before timeout expired, process sets the $test system variable to 1, otherwize to 0. If the /ATO option has not been specified, process does not change the $test value.

The *close* command options

TCP device does not accept any option on *close* command even it is specified, all options are ignored. All internal opened TCP sockets closed.

10.12 TNT

TNT device is a input-output device created automatically to handle input-output for telnet clients. Telnet client application or other with telnet protocol interact with MiniM process over TCP/IP connection. TNT device translates telnet protocol to device program interface.

TNT device creates automatically on MiniM process starts to handle incoming telnet connection. It is principal device and cannot be opened or

closed manually. Device identification string have a prefix with device type "|TNT|" with following integer. It is internal socket number and can be used by external functions, for example $zdll. Device identification string does not changes while process still active.

TNT devices does not support any *open* and *close* options.

The *use* command options

The *use* command with TNT device can accept the following options:

/COLUMNS = expr	Use this column count.
/LINES = expr	Use this lines count.
/ECHO = expr	Use or not echo mode on input.
/TERM = expr	Read string terminator.
/CTRLC = expr	Generate or not error on Ctrl+C pressing
/MODE = expr	Use text (t) or binary (b) mode to read bytes from telnet client
other	Ignored

The /COLUMNS = expr option

The /COLUMNS = expr specifies how many columns need to be used to display information. It is used by MiniM process. By default used 80 columns.

The /COLUMNS = expr option can be specified by position in 1 position and by name in any position.

The *expr* expression evaluates as an integer. If result is less than or equal 0, process generate the <DEVPARAM> error.

The /LINES = expr option

The /LINES = expr option specify how many lines need to be used to display data on the telnet screen. By default used 25 lines.

The /LINES = expr option can be specified by position in 2 position or by name in any position.

The *expr* expression evaluates as an integer. If result is less than or equal 0, process generate the <DEVPARAM> error.

Both /LINES and /COLUMNS options does not send any data to telnet client, they are used only by MiniM device to recalculate data visibility.

The /ECHO = expr option

This option specify echo mode while entering data in telnet client. MiniM process controls any input and resend characters back to client to display input result correctly. By default echo is enabled. If telnet client support own echo mode, it must be turned off.

The *expr* expression evaluates as an integer. If it is 0, echo is off, otherwize echo is on.

The /ECHO = expr option can be specified by position in 3 position or by name in any position.

The /CTRLC = expr option

The /CTRLC = expr option specify error generation behavior on Ctrl+C pressing. The *expr* expression evaluates as an integer. If it is 0, process does not generate <INTERRUPT> error on Ctrl+C pressing, otherwise generate. By default <INTERRUPT> error generating mode is enabled. This option can be specified by name only.

The /MODE = expr option

Option /MODE = expr switches device for read mode from telnet client from text to binary and back. Option use text (t or T) or binary (b or B) mode. By default telnet device use text mode. If device work in binary mode, Ctrl+C interruption turned off, but this setting still active and Ctrl+C interruption is restored by current device setting on return back to text mode.

The expr value evaluates as a string and option use only first symbol case insensitive. If this symbol is t or T, device use text mode and if symbol is b or B, device use binary mode.

If device work in binary mode, input string editing is turned off and no other input escape sequences are threated, all bytes goes to input string. Read character command places symbol was read into system variable $KEY and read string command places into system variable $KEY the string terminator if read string was terminated by terminator.

The /MODE = expr have not by-position specification and supports only named form in any position, the option name must be specified directly.

Example to read bytes in binary mode:

```
u $p:(/mode="b":/term="":/echo=0)
k ^CH
f i=1:1:16 r *ch s ^CH($i(^CH))=ch
u $p:(/mode="t":/echo=1)
zw ^CH
```

Example to read strings in binary mode with terminator of line feed character:

```
u $p:(/mode="b":/term=$c(10):/echo=0)
k ^STR
f i=1:1:5 r str s ^STR($i(^STR))=$zquote(str)
u $p:(/mode="t":/echo=1)
zw ^STR
```

The /TERM = expr option

The /TERM = expr option specify read string retminator. Read string terminates if process reads full specified terminator and terminator does not included into read string result. This option can be specified by name only. On top level commands input read string terminator does not used anyway and continues to use while commands executes.

Read string terminators

By default for TNT device are used 2 read string terminators - ENTER and ESCAPE symbols. This terminators are used by default and if specified empty string terminator and on top level commands input.

On top level command input (in text mode) MiniM process enables internal string editor and interact with telnet client by telnet protocol over tcp/ip connection.

MiniMono difference

MiniM Embedded Edition does not implement —TNT— device.

Chapter 11

Error Handling

11.1 Error handling tools

To handle errors MiniM Database Server implements standard and some extended functionality. System variables to use in error handling are:

$zerror	Shoe extended last error text for error occured.
$ecode	Contains list of last fired errors and assignment generate an error.
$estack	Show stack level counted from last made *new $estack* command or from top level.
$etrap	Contains error handling commands sequence.
$stack()	Returns information about process stack and execution states.

Extended MiniM Database Server error handling tools are functions $view("err"):

$view("err",1)	Return internal error place in MiniM source code. This information can help to localize an error reasons with MiniM Database server support.
$view("err",2,code)	Return extended error text description for specified standard error code which are implemented by MiniM Database Server.

Error handling control is made by assignment to system variables *$ecode* and *$etrap* and call a *new* command with *$estack* and *$etrap*. The *new*

command with the *$etrap* system variable can be used in extended assignable form. System variable *$zerror* is accessible to read only, cannot be changed directly and copy cannot be made by the *new* command.

The *break* command now is not implemented in MiniM Database Server. This command is allowed in routines but on execution in any form generates an error <UNIMPLEMENTED>.

Current MiniM Database Server implements only error handling methods and tools defined in current MUMPS standard as defined in ANSI MDC X11.1-1995, now withdrawn, and currently active ISO Standard MUMPS based on ANSI X11.

11.2 Error handler scheme

On error generation MiniM Database Server check the value of system variable $etrap on the current stack level. On new stack level creation by *do*, *xecute* commands or calling used function the $etrap variable got the value from previous stack level. If the *new* command was applied to $etrap variable, from this stack level and later this variable will have new assigned value and on stack level leaving got value on the previous stack level. After handling error on current stack level MiniM continues unroll stack and check the value of $etrap to execute error handler.

If the value of $etrap is not empty string, MiniM executes this code as MUMPS commands instead current stack level commands and without creating new stack level.

To specify place on the stack where need to handle errors programmer can use system variable $estack. On new stack creation this variable automatically increment value by one. If the *new* command was applied to the $estack value, this variable got the value of 0 and increments every new stack automatically. On return to previous stack level system variable $estack restores previous value. This behavior alow programmer to check stack level to execute error handler, for example:

```
n $es,$et="q:$es d err"
```

Here the *new* command applied to $estack variable and from this level value increments by one. For system variable $etrap was created copy of value where specified execute *d err* if the value of $estack is zero. On error

execution MiniM process unrolls stack to level where the value of $estack was zero and on this level will be executed command *d err*.

If on stack unroll MiniM process does not find nonempty value of $etrap, process continues stack unroll up to top stack level. If it is console or telnet process, control will be passed to the top command input and error will be displayed by default on the screen. If process is not interactive, process terminate execution and halts.

11.3 Error generation

To generate error there is possible three ways: 1) assign non empty string to system variable *$ecode*, 2) call operation which generate an error, for example divide by zero or call undefined variable and 3) call *ztrap* command.

In the second case be generated a special error, defined in the standard or extended and implemented by MiniM, and in first and third cases programmer can control the error content.

On *$ecode* assignment to an empty string the *$ecode* content clears and no other side effect are made. On assignment non empty string to *$ecode* process generate an error <ECODETRAP>.

On any standard error or MiniM extended error it is added as defined in MUMPS standard into *$ecode* value with comma as a delimiter. For example:

```
USER>s $ec="w"

<ECODETRAP>
USER>s $ec=$ec_"w"

<ECODETRAP>
USER>w $ec
ww
USER>w e

<UNDEFINED> *e
USER>w $ec
ww,M6,
```

In standard defined recommended rule to make error text. If error code starts from the "M" character, it is a standard error. If error code starts

from the "Z" character, it is implementation-specific extended error (MiniM extended error for MiniM Database Server). Otherwise, if error code starts from the "U" character, it is a user-defined error. This convention can differentiate error classes.

It is recommended to use some of defined rules to create errors in MUMPS applications to construct the *$ecode* value. It may be standard defined conventions or any other compatible and portable conventions.

Chapter 12

Regular Expressions

12.1 Regular Expressions Options

Functions *$zpcrematch()*, *$zpcresearch()* and *$zpcrereplace()* are built using PCRE library (http://www.pcre.org), author Philip Hazel. To use this functions in MiniM can be useful any documentation or samples code and libraries of regular expressions. Programmers who use other programming languages can apply knowledge and use the same regular expressions.

Regular expressions documentation is built using original documentation from http://www.pcre.ru and http://www.pcre.org and adapted for MUMPS language and MiniM specific implementation.

Value of the *options* argument, passed to functions *$zpcrematch()*, *$zpcresearch()* and *$zpcrereplace()*, must be a string with special symbol flags (case insensitive). If flag is present in the *options* argument, it is equal regular expression option is specified. If the *options* argument is omitted, it is equal passing empty string with no flags. If the *options* argument contains unsupported symbol, this symbol ignores.

See the tables how *options* symbols conforms original PCRE options:

A	Conforms to PCRE_ANCHORED PCRE option. If this bit is set, the pattern is forced to be "anchored", that is, it is constrained to match only at the first matching point in the string that is being searched (the "subject string"). For function *$zpcrematch()* this flag does not affect, because function make full string match.

I Conforms to PCRE_CASELESS PCRE option. If this bit is set, letters in the pattern match both upper and lower case letters. It is equivalent to Perl's /i option, and it can be changed within a pattern by a (?i) option setting.

D Conforms PCRE_DOLLAR_ENDONLY PCRE option. If this bit is set, a dollar metacharacter in the pattern matches only at the end of the subject string. Without this option, a dollar also matches immediately before a newline at the end of the string (but not before any other newlines). The D flag is ignored if M flag is set. There is no equivalent to this option in Perl, and no way to set it within a pattern.

S Conforms to PCRE_DOTALL PCRE option. If this bit is set, a dot metacharater in the pattern matches all char- acters, including those that indicate newline. Without it, a dot does not match when the current position is at a newline. This option is equivalent to Perl's /s option, and it can be changed within a pattern by a (?s) option setting. A negative class such as [^a] always matches newline characters, independent of the setting of this option.

M Conforms to PCRE_MULTILINE PCRE option. By default, PCRE treats the subject string as consisting of a single line of characters (even if it actually contains newlines). The "start of line" metacharacter (^) matches only at the start of the string, while the "end of line" metacharacter ($) matches only at the end of the string, or before a terminating newline (unless D flag is set). This is the same as Perl.

When M flag it is set, the "start of line" and "end of line" constructs match immediately following or immediately before internal newlines in the subject string, respectively, as well as at the very start and end. This is equivalent to Perl's /m option, and it can be changed within a pattern by a (?m) option setting. If there are no newlines in a subject string, or no occurrences of ^ or $ in a pattern, setting M flag has no effect.

For function $zpcrematch() this flag have no effect because function match all string.

U Conforms to PCRE_UNGREEDY PCRE option. This option inverts the "greediness" of the quantifiers so that they are not greedy by default, but become greedy if followed by "?". It is not compatible with Perl. It can also be set by a (?U) option setting within the pattern.

X Conforms to PCRE_EXTENDED PCRE option. If this flag is set, whitespace data characters in the pattern are totally ignored except when escaped or inside a character class. Whitespace does not include the VT character (code 11). In addition, characters between an unescaped # outside a character class and the next newline, inclusive, are also ignored. This is equivalent to Perl's /x option, and it can be changed within a pattern by a (?x) option setting.

This option makes it possible to include comments inside complicated patterns. Note, however, that this applies only to data characters. Whitespace characters may never appear within special character sequences in a pattern, for example within the sequence (?(which introduces a conditional subpattern.

T Conforms to PCRE_EXTRA PCRE option. This option was invented in order to turn on additional functionality of PCRE that is incompatible with Perl, but it is currently of very little use. When set, any backslash in a pattern that is followed by a letter that has no special meaning causes an error, thus reserving these combinations for future expansion. By default, as in Perl, a backslash followed by a letter with no special meaning is treated as a literal. (Perl can, however, be persuaded to give an error for this, by running it with the -w option.) There are at present no other features controlled by this option. It can also be set by a (?X) option setting within a pattern.

G Execute global search or replace, all occurences.

B Conforms to PCRE_NOTBOL PCRE option. This option specifies that first character of the subject string is not the beginning of a line, so the circumflex metacharacter should not match before it. Setting this without M flag causes circumflex never to match. This option affects only the behaviour of the circumflex metacharacter. It does not affect \A.

E Conforms to PCRE_NOTEOL PCRE option. This option specifies that the end of the subject string is not the end of a line, so the dollar metacharacter should not match it nor (except in multiline mode) a newline immediately before it. Setting this without M flag causes dollar never to match. This option affects only the behaviour of the dollar metacharacter. It does not affect \Z or \z.

P Argument to replace *with* for function *$zpcrereplace()* is used as string with pseudovariables.

12.2 Regular Expressions Syntax

The PCRE library is a set of functions (*$zpcrematch()*, *$zpcresearch()* and *$zpcrereplace()*) that implement regular expression pattern matching using the same syntax and semantics as Perl 5, with just a few differences.

Differences from Perl

Differences are listed in comparision with Perl 5.005.

1 Space symbols in PCRE functions are the same as space symbols in MiniM. It is space, horizontal tabulation, carriage return, line feed, vertical tabulation and non-breakable space.

2 PCRE does not allow repeat quantifiers on lookahead assertions. Perl permits them, but they do not mean what you might think. For example, (?!a)3 does not assert that the next three characters are not "a". It just asserts that the next character is not "a" three times.

3 Capturing subpatterns that occur inside negative lookahead asser tions are counted, but their entries in the offsets vector are never set. Perl sets its numerical variables from any such patterns that are matched before the assertion fails to match something (thereby succeed ing), but only if the negative lookahead assertion contains just one branch.

4 Though binary zero characters are supported in the subject string, they are not allowed in a pattern string because it is passed as a normal C string, terminated by zero. The escape sequence \0 can be used in the pattern to represent a binary zero.

5 The following Perl escape sequences are not supported: \l, \u, \L, \U, and \N. In fact these are implemented by Perl's general string-handling and are not part of its pattern matching engine. If any of these are encountered by PCRE, an error is generated.

6 Perl escape sequence \G does not supported.

7 PCRE does not support sequence as pattern(?code).

8 Pattern "ˆ(a)?(?(1)a—b)+$" conforms to string 'a' in PERL, but not in PCRE. In the same time pattern "ˆ(a)?a" conforms to string 'a' in Perl and in PCRE, leaves variable $1 unset.

10 PCRE provides some extensions to the Perl regular expression facilities:

(a) Although lookbehind assertions in PCRE must match fixed length strings, each alternative branch of a lookbehind assertion can match a different length of string. Perl requires them all to have the same length.

(b) If D flag (PCRE_DOLLAR_ENDONLY) is set and M flag (PCRE_MULTILINE) is not set, the $ metacharacter matches only at the very end of the string.

(c) If T flag (PCRE_EXTRA) is set, a backslash followed by a letter with no special meaning is faulted. Otherwise, like Perl, the backslash is quietly ignored.

(d) If U flag (PCRE_UNGREEDY) is set, the greediness of the repetition quantifiers is inverted, that is, by default they are not greedy, but if followed by a question mark they are.

Following is described Perl-compatible regular expressions syntax (PCRE). Regular expressions are very good described in Perl documentation and in very much books with many samples, for example, "Mastering Regular Expressions", written by Effrey Friedl's published by O'Reilly (ISBN 1-56592-257-3).

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern *The quick brown fox* matches a portion of a subject string that is identical to itself.

Meta-characters

The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of *meta-characters* , which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of meta-characters: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized in square brackets. Outside square brackets, the meta-characters are as follows:

\	general escape character with several uses
^	assert start of subject (or line, in multiline mode)
$	assert end of subject (or line, in multiline mode)
.	(dot) match any character except newline (by default)
[	start character class definition
]	end character class definition
|	start of alternative branch
(	start subpattern
)	end subpattern
?	extends the meaning of (, also 0 or 1 quantifier, also quantifier minimizer
*	0 or more quantifier
+	1 or more quantifier
{	start min/max quantifier
}	end min/max quantifier

Part of a pattern that is in square brackets is called a "character class". In a character class the only meta-characters are:

\	general escape character

^	negate the class, but only if the first character
-	indicates character range
]	terminates the character class

The following sections describe the use of each of the meta-characters.

Backslash

The backslash (\) character has several uses. Firstly, if it is followed by a non-alphanumeric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, if you want to match a "*" character, you write "*" in the pattern. This applies whether or not the following character would otherwise be interpreted as a meta-character, so it is always safe to precede a non-alphanumeric with "\" to specify that it stands for itself. In particular, if you want to match a backslash, you write "\\".

If a pattern is compiled with the X (PCRE_EXTENDED) option, whitespace in the pattern (other than in a character class) and characters between a "#" outside a character class and the next newline character are ignored. An escaping backslash can be used to include a whitespace or "#" character as part of the pattern.

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is usually easier to use one of the following escape sequences than the binary character it represents:

\a	alarm, that is, the BEL character (hex 07)
\cx	"control-x", where x is any character
\e	escape (hex 1B)
\f	formfeed (hex 0C)
\n	newline (hex 0A)
\r	carriage return (hex 0D)
\t	tab (hex 09)
\xhh	character with hex code hh
\ddd	character with octal code ddd, or backreference

The precise effect of "\cx " is as follows: if " x " is a lower case letter, it

is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus "\cz " becomes hex 1A, but "\c " becomes hex 3B, while "\c; " becomes hex 7B.

After "\x ", up to two hexadecimal digits are read (letters can be in upper or lower case).

After "\0 " up to two further octal digits are read. In both cases, if there are fewer than two digits, just those that are present are used. Thus the sequence "\0\x\07 " specifies two binary zeros followed by a BEL character. Make sure you supply two digits after the initial zero if the character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a back reference . A description of how this works is given later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE re-reads up to three octal digits following the backslash, and generates a single byte from the least significant 8 bits of the value. Any subsequent digits stand for themselves. For example:

\040	is another way of writing a space
\40	is the same, provided there are fewer than 40 previous capturing subpatterns
\7	is always a back reference
\11	might be a back reference, or another way of writing a tab
\011	is always a tab
\0113	is a tab followed by the character "3"
\113	is the character with octal code 113 (since there can be no more than 99 back references)
\377	is a byte consisting entirely of 1 bits
\81	is either a back reference, or a binary zero followed by the two characters "8" and "1"

Note that octal values of 100 or greater must not be introduced by a leading zero, because no more than three octal digits are ever read.

All the sequences that define a single byte value can be used both inside

and outside character classes. In addition, inside a character class, the sequence "\b" is interpreted as the backspace character (hex 08). Outside a character class it has a different meaning (see below).

The third use of backslash is for specifying generic character types:

\d	any decimal digit
\D	any character that is not a decimal digit
\s	any whitespace character
\S	any character that is not a whitespace character
\w	any "word" character
\W	any "non-word" character

Each pair of escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair.

A "word" character is any letter or digit or the underscore character, that is, any character which can be part of a Perl "word".

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, since there is no character to match.

The fourth use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match, without consuming any characters from the subject string. The use of subpatterns for more complicated assertions is described below. The backslashed assertions are:

\b	word boundary
\B	not a word boundary
\A	start of subject (independent of multiline mode)
\Z	end of subject or newline at end (independent of multiline mode)
\z	end of subject (independent of multiline mode)

These assertions may not appear in character classes (but note that "\b" has a different meaning, namely the backspace character, inside a character class).

A word boundary is a position in the subject string where the current character and the previous character do not both match \w or \W (i.e. one

matches \w and the other matches \W), or the start or end of the string if the first or last character matches \w , respectively.

The \A, \Z, and \z assertions differ from the traditional circumflex and dollar (described below) in that they only ever match at the very start and end of the subject string, whatever options are set. They are not affected by the M flag (PCRE_MULTILINE) or D flag (PCRE_DOLLAR_ENDONLY) options. The difference between \Z and \z is that \Z matches before a newline that is the last character of the string as well as at the end of the string, whereas \z matches only at the end.

Circumflex and dollar

Outside a character class, in the default matching mode, the circumflex character (^) is an assertion which is true only if the current matching point is at the start of the subject string. Inside a character class, circumflex has an entirely different meaning (see below).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative in which it appears if the pattern is ever to match that branch. If all possible alternatives start with a circumflex, that is, if the pattern is constrained to match only at the start of the subject, it is said to be an "anchored" pattern. (There are also other constructs that can cause a pattern to be anchored.)

A dollar character is an assertion which is TRUE only if the current matching point is at the end of the subject string, or immediately before a newline character that is the last character in the string (by default). Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. Dollar has no special meaning in a character class.

The meaning of dollar can be changed so that it matches only at the very end of the string, by setting the "D" (PCRE_DOLLAR_ENDONLY) option at compile or matching time. This does not affect the \Z assertion.

The meanings of the circumflex and dollar characters are changed if the "M" (PCRE_MULTILINE) option is set. When this is the case, they match immediately after and immediately before an internal "\n" character, respectively, in addition to matching at the start and end of the subject string. For example, the pattern /^abc$/ matches the subject string "def\nabc" in multiline mode, but not otherwise. Consequently, patterns that are anchored in single line mode because all branches start with "^" are not anchored in multiline mode. The "D" (PCRE_DOLLAR_ENDONLY) option is ignored if "M" (PCRE_MULTILINE) is set.

Note that the sequences \A, \Z, and \z can be used to match the start and end of the subject in both modes, and if all branches of a pattern start with \A is it always anchored, whether "M" (PCRE_MULTILINE) is set or not.

Full stop

Outside a character class, a dot in the pattern matches any one character in the subject, including a non-printing character, but not (by default) newline. If the "S" (PCRE_DOTALL) option is set, then dots match newlines as well. The handling of dot is entirely independent of the handling of circumflex and dollar, the only relationship being that they both involve newline characters. Dot has no special meaning in a character class.

Square brackets

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special. If a closing square bracket is required as a member of the class, it should be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the subject; the character must be in the set of characters defined by the class, unless the first character in the class is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class [aeiou] matches any lower case vowel, while [^aeiou] matches any character that is not a lower case vowel. Note that a circumflex is just a convenient notation for specifying the characters which are in the class by enumerating those that are not. It is not an assertion: it still consumes a character from the subject string, and fails if the current pointer is at the end of the string.

When caseless matching is set, any letters in a class represent both their upper case and lower case versions, so for example, a caseless [aeiou] matches "A" as well as "a", and a caseless [^aeiou] does not match "A", whereas a caseful version would.

The newline character is never treated in any special way in character classes, whatever the setting of the "S" (PCRE_DOTALL) or "M" (PCRE_MULTILINE) is. A class such as [^a] will always match a newline.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, [d-m] matches any letter between d and

m, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

It is not possible to have the literal character "]" as the end character of a range. A pattern such as [W-]46] is interpreted as a class of two characters ("W" and "-") followed by a literal string "46]", so it would match "W46]" or "-46]". However, if the "]" is escaped with a backslash it is interpreted as the end of range, so [W-\]46] is interpreted as a single class containing a range followed by two separate characters. The octal or hexadecimal representation of "]" can also be used to end a range.

Ranges operate in ASCII collating sequence. They can also be used for characters specified numerically, for example [\000-\037]. If a range that includes letters is used when caseless matching is set, it matches the letters in either case. For example, [W-c] is equivalent to [][\ˆ_‘wxyzabc], matched caselessly, and if character tables for the "fr" locale are in use, [\xc8-\xcb] matches accented E characters in both cases.

The character types \d, \D, \s, \S, \w, and \W may also appear in a character class, and add the characters that they match to the class. For example, [\dABCDEF] matches any hexadecimal digit. A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class [ˆ\W_] matches any letter or digit, but not underscore.

All non-alphanumeric characters other than \, -, ˆ (at the start) and the terminating] are non-special in character classes, but it does no harm if they are escaped.

Vertical bar

Vertical bar "|" characters are used to separate alternative patterns. For example, the pattern gilbert|sullivan matches either "gilbert" or "sullivan". Any number of alternatives may appear, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern (defined below), "succeeds" means matching the rest of the main pattern as well as the alternative in the subpattern.

Internal option setting

The settings of "I" (PCRE_CASELESS), "M" (PCRE_MULTILINE), "S" (PCRE_DOTALL), "U" (PCRE_UNGREEDY), "T" (PCRE_EXTRA), and "X" (PCRE_EXTENDED) can be changed from within the pattern by a

sequence of Perl option letters enclosed between "(?" and ")". The option letters are:

i	for "I" (PCRE_CASELESS)
m	for "M" (PCRE_MULTILINE)
s	for "S" (PCRE_DOTALL)
x	for "X" (PCRE_EXTENDED)
U	for "U" (PCRE_UNGREEDY)
X	for "T" (PCRE_EXTRA)

For example, (?im) sets caseless, multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and a combined setting and unsetting such as (?im-sx), which sets "I" (PCRE_CASELESS) and "M" (PCRE_MULTILINE) while unsetting "S" (PCRE_DOTALL) and "X" (PCRE_EXTENDED), is also permitted. If a letter appears both before and after the hyphen, the option is unset.

When an option change occurs at top level (that is, not inside subpattern parentheses), the change applies to the remainder of the pattern that follows. So /ab(?i)c/ matches only "abc" and "abC".

If an option change occurs inside a subpattern, the effect is different. This is a change of behaviour in Perl 5.005. An option change inside a subpattern affects only that part of the subpattern that follows it, so (a(?i)b)c matches abc and aBc and no other strings (assuming "I" (PCRE_CASELESS) is not used). By this means, options can be made to have different settings in different parts of the pattern. Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example, (a(?i)b|c) matches "ab", "aB", "c", and "C", even though when matching "C" the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time. There would be some very weird behaviour otherwise.

The PCRE-specific options "U" (PCRE_UNGREEDY) and "T" (PCRE_EXTRA) can be changed in the same way as the Perl-compatible options by using the characters U and X respectively. The (?X) flag setting is special in that it must always occur earlier in the pattern than any of the additional features it turns on, even when it is at top level. It is best put at the start.

Subpatterns

Subpatterns are delimited by parentheses (round brackets), which can be nested. Marking part of a pattern as a subpattern does two things:

1 It localizes a set of alternatives. For example, the pattern cat(aract|erpillar|) matches one of the words "cat", "cataract", or "caterpillar". Without the parentheses, it would match "cataract", "erpillar" or the empty string.

2 It sets up the subpattern as a capturing subpattern (as defined above). Opening parentheses are counted from left to right (starting from 1) to obtain the numbers of the capturing subpatterns.

For example, if the string "the red king" is matched against the pattern the ((red|white) (king|queen)) the captured substrings are "red king", "red", and "king", and are numbered 1, 2, and 3.

The fact that plain parentheses fulfil two functions is not always helpful. There are often times when a grouping subpattern is required without a capturing requirement. If an opening parenthesis is followed by "?:", the subpattern does not do any capturing, and is not counted when computing the number of any subsequent capturing subpatterns. For example, if the string "the white queen" is matched against the pattern the ((?:red|white) (king|queen)) the captured substrings are "white queen" and "queen", and are numbered 1 and 2. The maximum number of captured substrings is 99, and the maximum number of all subpatterns, both capturing and non-capturing, is 200.

As a convenient shorthand, if any option settings are required at the start of a non-capturing subpattern, the option letters may appear between the "?" and the ":". Thus the two patterns

(?i:saturday|sunday)
(?:(?i)saturday|sunday)

match exactly the same set of strings. Because alternative branches are tried from left to right, and options are not reset until the end of the subpattern is reached, an option setting in one branch does affect subsequent branches, so the above patterns match "SUNDAY" as well as "Saturday".

Repetition

Repetition is specified by quantifiers, which can follow any of the following items:

- a single character, possibly escaped;
- the . metacharacter;

- a character class;
- a back reference (see next section);
- a parenthesized subpattern (unless it is an assertion - see below).

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in curly brackets (braces), separated by a comma. The numbers must be less than 65536, and the first must be less than or equal to the second. For example: z2,4 matches "zz", "zzz", or "zzzz". A closing brace on its own is not a special character. If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus [aeiou]3, matches at least 3 successive vowels, but may match many more, while \d8 matches exactly 8 digits. An opening curly bracket that appears in a position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, ,6 is not a quantifier, but a literal string of four characters.

The quantifier 0 is permitted, causing the expression to behave as if the previous item and the quantifier were not present.

For convenience (and historical compatibility) the three most common quantifiers have single-character abbreviations:

$*$	equivalent to {0,}
$+$	equivalent to {1,}
?	equivalent to {0,1}

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example: (a?)*

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

By default, the quantifiers are "greedy", that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between

the sequences /* and */ and within the sequence, individual * and / characters may appear. An attempt to match C comments by applying the pattern /*.**/ to the string /* first command */ not comment /* second comment */ fails, because it matches the entire string due to the greediness of the .* item.

However, if a quantifier is followed by a question mark, then it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern /*.*?*/ does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in \d??\d which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the "U" (PCRE_UNGREEDY) option is set (an option which is not available in Perl) then the quantifiers are not greedy by default, but individual ones can be made greedy by following them with a question mark. In other words, it inverts the default behaviour.

Quantifiers followed by + are "possessive". They eat as many characters as possible and don't return to match the rest of the pattern. Thus .*abc matches "aabc" but .*+abc doesn't because .*+ eats the whole string.

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more store is required for the compiled pattern, in proportion to the size of the minimum or maximum.

If a pattern starts with .* or .0, and the "S" (PCRE_DOTALL) option (equivalent to Perl's /s) is set, thus allowing the . to match newlines, then the pattern is implicitly anchored, because whatever follows will be tried against every character position in the subject string, so there is no point in retrying the overall match at any position after the first. PCRE treats such a pattern as though it were preceded by \A. In cases where it is known that the subject string contains no newlines, it is worth setting "S" (PCRE_DOTALL) when the pattern begins with .* in order to obtain this optimization, or alternatively using '^' to indicate anchoring explicitly.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For example, after (tweedle[dume]3\s*)+ has matched "tweedledum tweedledee" the value of the captured substring is "tweedledee". However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations. For

example, after /(a|(b))+/ matches "aba" the value of the second captured substring is "b".

Back references

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a back reference to a capturing subpattern earlier (i.e. to its left) in the pattern, provided there have been that many previous capturing left parentheses.

However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. See the section entitled "Backslash" above for further details of the handling of digits following a backslash.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself. So the pattern (sens|respons)e and \1ibility matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If caseful matching is in force at the time of the back reference, then the case of letters is relevant. For example, ((?i)rah)\s+\1 matches "rah rah" and "RAH RAH", but not "RAH rah", even though the original capturing subpattern is matched caselessly.

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, then any back references to it always fail. For example, the pattern (a—(bc))\2 always fails if it starts to match "a" rather than "bc". Because there may be up to 99 back references, all digits following the backslash are taken as part of a potential back reference number. If the pattern continues with a digit character, then some delimiter must be used to terminate the back reference. If the "X" (PCRE_EXTENDED) option is set, this can be whitespace. Otherwise an empty comment can be used.

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, (a\1) never matches. However, such references can be useful inside repeated subpatterns. For example, the pattern (a|b\1)+ matches any number of "a"s and also "aba", "ababaa" etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

Assertions

An assertion is a test on the characters following or preceding the current matching point that does not actually consume any characters. The simple assertions coded as \b, \B, \A, \Z, \z, ^ and $ are described above. More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it.

An assertion subpattern is matched in the normal way, except that it does not cause the current matching position to be changed. Lookahead assertions start with (?= for positive assertions and (?! for negative assertions. For example, \w+(?=;) matches a word followed by a semicolon, but does not include the semicolon in the match, and foo(?!bar) matches any occurrence of "foo" that is not followed by "bar". Note that the apparently similar pattern (?!foo)bar does not find an occurrence of "bar" that is preceded by something other than "foo"; it finds any occurrence of "bar" whatsoever, because the assertion (?!foo) is always TRUE when the next three characters are "bar". A lookbehind assertion is needed to achieve this effect.

Lookbehind assertions start with (?¡= for positive assertions and (?¡! for negative assertions. For example, (?¡!foo)bar does find an occurrence of "bar" that is not preceded by "foo". The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are several alternatives, they do not all have to have the same fixed length. Thus (?¡=bullock|donkey) is permitted, but (?¡!dogs?|cats?) causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl 5.005, which requires all branches to match the same length of string. An assertion such as (?¡=ab(c|de)) is not permitted, because its single top-level branch can match two different lengths, but it is acceptable if rewritten to use two top-level branches: (?¡=abc|abde) The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed width and then try to match. If there are insufficient characters before the current position, the match is deemed to fail. Lookbehinds in conjunction with once-only subpatterns can be particularly useful for matching at the ends of strings; an example is given at the end of the section on once-only subpatterns.

Several assertions (of any sort) may occur in succession. For example, (?¡=\d3)(?¡!999)foo matches "foo" preceded by three digits that are not "999". Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous

three characters are all digits, then there is a check that the same three characters are not "999". This pattern does not match "foo" preceded by six characters, the first of which are digits and the last three of which are not "999". For example, it doesn't match "123abcfoo". A pattern to do that is (?¡=\d3...)(?¡!999)foo. This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not "999".

Assertions can be nested in any combination. For example, (?¡=(?¡!foo)bar)baz matches an occurrence of "baz" that is preceded by "bar" which in turn is not preceded by "foo", while (?¡=\d3...(?¡!999))foo is another pattern which matches "foo" preceded by three digits and any three characters that are not "999".

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

Assertions count towards the maximum of 200 parenthesized subpatterns.

Once-only subpatterns

With both maximizing and minimizing repetition, failure of what follows normally causes the repeated item to be re-evaluated to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it fail earlier than it otherwise might, when the author of the pattern knows there is no point in carrying on.

Consider, for example, the pattern \d+foo when applied to the subject line 123456bar.

After matching all 6 digits and then failing to match "foo", the normal action of the matcher is to try again with only 5 digits matching the \d+ item, and then with 4, and so on, before ultimately failing. Once-only subpatterns provide the means for specifying that once a portion of the pattern has matched, it is not to be re-evaluated in this way, so the matcher would give up immediately on failing to match "foo" the first time. The notation is another kind of special parenthesis, starting with (?¿ as in this example:

(?¿\d+)bar

This kind of parenthesis "locks up" the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from

backtracking into it. Backtracking past it to previous items, however, works as normal.

An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point in the subject string.

Once-only subpatterns are not capturing subpatterns. Simple cases such as the above example can be thought of as a maximizing repeat that must swallow everything it can. So, while both \d+ and \d+? are prepared to adjust the number of digits they match in order to make the rest of the pattern match, (?¿\d+) can only match an entire sequence of digits.

This construction can of course contain arbitrarily complicated subpatterns, and it can be nested.

Once-only subpatterns can be used in conjunction with look-behind assertions to specify efficient matching at the end of the subject string. Consider a simple pattern such as abcd$ when applied to a long string which does not match. Because matching proceeds from left to right, PCRE will look for each "a" in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as ˆ.*abcd$ then the initial .* matches the entire string at first, but when this fails (because there is no following "a"), it backtracks to match all but the last character, then all but the last two characters, and so on. Once again the search for "a" covers the entire string, from right to left, so we are no better off. However, if the pattern is written as ˆ(?¿.*)(?¡=abcd) then there can be no backtracking for the .* item; it can match only the entire string. The subsequent lookbehind assertion does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

When a pattern contains an unlimited repeat inside a subpattern that can itself be repeated an unlimited number of times, the use of a once-only subpattern is the only way to avoid some failing matches taking a very long time indeed. The pattern (\D+|¡\d+¿)*[!?] matches an unlimited number of substrings that either consist of non-digits, or digits enclosed in <>, followed by either ! or ?. When it matches, it runs quickly. However, if it is applied to aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa it takes a long time before reporting failure. This is because the string can be divided between the two repeats in a large number of ways, and all have to be tried. (The example used [!?] rather than a single character at the end, because both PCRE and Perl have an optimization that allows for fast failure when a single character is used. They remember the last single character that is

required for a match, and fail early if it is not present in the string.) If the pattern is changed to ((?¿\D+)|¡\d+¿)*[!?] sequences of non-digits cannot be broken, and failure happens quickly.

Conditional subpatterns

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a previous capturing subpattern matched or not. The two possible forms of conditional subpattern are:

(?(condition)yes-pattern)
(?(condition)yes-pattern—no-pattern)

If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs.

There are two kinds of condition. If the text between the parentheses consists of a sequence of digits, then the condition is satisfied if the capturing subpattern of that number has previously matched. Consider the following pattern, which contains non-significant white space to make it more readable (assume the "X" (PCRE_EXTENDED) option) and to divide it into three parts for ease of discussion: (\( )? [^()]+ (?(1) \))

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did, that is, if subject started with an opening parenthesis, the condition is TRUE , and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, since no-pattern is not present, the subpattern matches nothing. In other words, this pattern matches a sequence of non-parentheses, optionally enclosed in parentheses.

If the condition is the string (R) , it is satisfied if a recursive call to the pattern or subpattern has been made. At "top level", the condition is false.

If the condition is not a sequence of digits or (R), it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again containing non-significant white space, and with the two alternatives on the second line:

```
(?(?=[^a-z]*[a-z])
\d{2}-[a-z]{3}-\d{2} | \d{2}-\d{2}-\d{2} )
```

The condition is a positive lookahead assertion that matches an optional sequence of non-letters followed by a letter. In other words, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative; otherwise it is matched against the second. This pattern matches strings in one of the two forms dd-aaa-dd or dd-dd-dd, where aaa are letters and dd are digits.

Comments

The sequence (?# marks the start of a comment which continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

If the "X" (PCRE_EXTENDED) option is set, an unescaped '#' character outside a character class introduces a comment that continues up to the next newline character in the pattern.

Recursive patterns

Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth. Perl 5.6 has provided an experimental facility that allows regular expressions to recurse (among other things). The special item (?R) is provided for the specific case of recursion. This PCRE pattern solves the parentheses problem (assume the "X" (PCRE_EXTENDED) option is set so that white space is ignored):
\(((?¿[^()]+) | (?R))* \)

First it matches an opening parenthesis. Then it matches any number of substrings which can either be a sequence of non-parentheses, or a recursive match of the pattern itself (i.e. a correctly parenthesized substring). Finally there is a closing parenthesis.

This particular example pattern contains nested unlimited repeats, and so the use of a once-only subpattern for matching strings of non-parentheses is important when applying the pattern to strings that do not match. For example, when it is applied to (aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa() it yields "no match" quickly. However, if a once-only subpattern is not used, the match runs for a very long time indeed because there are so many different ways the + and * repeats can carve up the subject, and all have to be tested before failure can be reported.

The values set for any capturing subpatterns are those from the outermost level of the recursion at which the subpattern value is set. If the pattern above

is matched against (ab(cd)ef) the value for the capturing parentheses is "ef", which is the last value taken on at the top level. If additional parentheses are added, giving \((((?¿[^()]+) | (?R))*) \) then the string they capture is "ab(cd)ef", the contents of the top level parentheses. If there are more than 15 capturing parentheses in a pattern, PCRE has to obtain extra memory to store data during a recursion. If no memory can be obtained, it saves data for the first 15 capturing parentheses only, as there is no way to give an out-of-memory error from within a recursion.

If the syntax for a recursive subpattern reference (either by number or by name) is used outside the parentheses to which it refers, it operates like a subroutine in a programming language. An earlier example pointed out that the pattern (sens|respons)e and \1ibility matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If instead the pattern (sens|respons)e and (?1)ibility is used, it does match "sense and responsibility" as well as the other two strings. Such references must, however, follow the subpattern to which they refer.

Performances

Certain items that may appear in patterns are more efficient than others. It is more efficient to use a character class like [aeiou] than a set of alternatives such as (a|e|i|o|u). In general, the simplest construction that provides the required behaviour is usually the most efficient. Jeffrey Friedl's book contains a lot of discussion about optimizing regular expressions for efficient performance.

When a pattern begins with .* and the "S" (PCRE_DOTALL) option is set, the pattern is implicitly anchored by PCRE, since it can match only at the start of a subject string. However, if "S" (PCRE_DOTALL) is not set, PCRE cannot make this optimization, because the . metacharacter does not then match a newline, and if the subject string contains newlines, the pattern may match from the character immediately following one of them instead of from the very start. For example, the pattern (.*) second matches the subject "first\nand second" (where \n stands for a newline character) with the first captured substring being "and". In order to do this, PCRE has to retry the match starting after every newline in the subject.

If you are using such a pattern with subject strings that do not contain newlines, the best performance is obtained by setting "S" (PCRE_DOTALL), or starting the pattern with ^.* to indicate explicit anchoring. That saves PCRE from having to scan along the subject looking for a newline to restart at.

Beware of patterns that contain nested indefinite repeats. These can take a long time to run when applied to a string that does not match. Consider the pattern fragment (a+)*

This can match "aaaa" in 33 different ways, and this number increases very rapidly as the string gets longer. (The * repeat can match 0, 1, 2, 3, or 4 times, and for each of those cases other than 0, the + repeats can match different numbers of times.) When the remainder of the pattern is such that the entire match is going to fail, PCRE has in principle to try every possible variation, and this can take an extremely long time.

An optimization catches some of the more simple cases such as (a+)*b where a literal character follows. Before embarking on the standard matching procedure, PCRE checks that there is a "b" later in the subject string, and if there is not, it fails the match immediately. However, when there is no following literal this optimization cannot be used. You can see the difference by comparing the behaviour of (a+)*\d with the pattern above. The former gives a failure almost instantly when applied to a whole line of "a" characters, whereas the latter takes an appreciable time with strings longer than about 20 characters.

Chapter 13

Errors List

13.1 MDC MUMPS standard errors list

MUMPS standard defines list of some errors. If this errors occurs, the appropriate error code with the "M" prefix written to the $ecode system variable. Here "M" prefix mean standard error code.

MiniM Database Server implements function $v("err",2) to return text description of error code.

```
$v("err",2,errcode)
```

Here instead *errcode* must be passed code of the error occurs, for example:

```
TEMP>s $ec=""

TEMP>w a

<UNDEFINED>
TEMP>s errcode=$p($ec,",",$l($ec,",")-1)

TEMP>s $e(errcode)=""

TEMP>w $v("err",2,errcode)
Undefined local variable.
```

MUMPS standard errors supported by MiniM Database Server:

1. Naked indicator undefined.
2. Invalid $FNumber code string combination.

3. $Random argument less than 1.
4. No true condition in $Select.
5. Line reference less than 0 (zero).
6. Undefined local variable.
7. Undefined global variable.
8. Undefined special variable.
9. Divide by zero.
10. Invalid pattern match range.
11. No parameters passed.
12. Invalid line reference (negative offset).
13. Invalid line reference (line not found).
14. Line level not one (1).
15. Undefined index variable.
16. Quit with an argument not allowed.
17. Quit with an argument required.
18. Fixed-length Read not greater than 0 (zero).
19. Cannot merge a tree or subtree into itself.
20. Line must have a formal list.
21. Formal list name duplication.
22. Set or Kill to ^$Global structured system variable name (SSVN) when data in global.
23. Set or Kill to ^$Job structured system variable name (SSVN) for non-existent job number.
24. Change to collation algorithm while subscripted local variables defined.
26. Non-existent environment (non-existent namespace).
27. Attempt to roll back a transaction that is not re-startable.
28. Mathematical function, parameter out of range.
29. Set or Kill on structured system variable name (SSVN) not allowed by implementation.
30. Reference to global variable with different collating sequence within a collating algorithm.
31. Device control mnemonic expression used for a device without a mnemonic space being selected.
32. Device control mnemonic used in user-defined mnemonic space which has no associated line.
33. Set or Kill to ^$Routine when the routine specified exists.
35. Device does not support mnemonic spaces.
36. Incompatible mnemonic spaces.
37. Read from device identified by null string.
38. Invalid structured system variable name (SSVN) subscript.

39. Invalid $Name argument.
40. Call by reference in the actual parameter list in Job command.
41. Invalid Lock argument within a transaction.
42. Invalid Quit within a transaction.
43. Invalid range value ($X,$Y).
44. Invalid command outside a transaction.
45. Invalid Goto reference.
57. A label is defined more than once in a routine.
58. Too few formal parameters.

13.2 MiniM errors list

When error in MiniM process occurs, error name writes into system variable $zerror between angle brackets with additional possible extended information to detail what happened. And value of this variable is displayed on the screen in interactive console and telnet screen. For example:

```
USER>w a

<UNDEFINED> *a

USER>w $ze
<UNDEFINED> *a
USER>
```

Some MiniM errors are standard errors and other are extended or user-generated.

MiniM supported errors list:

BYTECODE

Routine bytecode corrupted.

Execution commands line bytecode corrupted.

Invalid or unsupported bytecode version.

COMMAND

Command got unsupported argument values, for example, *write *expr*, where *expr* - negative unsupported integer.

The *quit* command with argument execution in context without return value ($quit=0).

The *quit* command with argument cannot break *for* command.

The *quit* command with argument cannot break *xecute* command.

The *kill* command cannot be applied to specified structured system variable.

The *lock* command contain invalid local or global variable name.

The *tcommit* command cannot be executed without transaction context ($tlevel=0).

The *merge* command cannot merge varible itself.

The *merge* command can merge only local or global variables.

The *zwrite* command cannot be applied to specified variable, it must be a local or global variable.

The *job* command to run child process with concurrent socket got invalid TCP device name.

DATABASE

Call undefined database name.

Database is defined, but don't mounted.

Process cannot find global start point.

One of data file or data file directory does not exist.

Failed to open data file to read.

Data file have unsupported size, file corrupted.

Error while write to data file.

Low level data file locking or unlocking failed.

Database cache corrupted.

System database %SYS is absent.

Temporary database TEMP is absent.

Data file block corrupted.

Error in cache flush to data file.

Data file extending failed.

Joutnal file write failed.

Database synchronization objects corrupted.

DBEXTEND

Database extend daemon failed to extend one of data file.

DBLIMIT

Process requires to extend database but detected database limit. Extending cannot be made now.

DBREADONLY

Process reguires write to readonly database.

DEVICE

Empty device identification string is unsupported.

Unknown device type specified in device identification string.

Error while interoperate with hardware or device driver.

Device operation not supported.

Device definition structure corrupted.

DEVICELIMIT

No more space in device table to open one more device.

DEVPARAM

Device option have unsupported value.

Mandatory device option value is absent.

DIVIDE

Division by zero is not supported.

EDITED

Routine bytecode is used for execution but is changed by other process.

ECODETRAP

Error generated by $ecode assignment non empty string.

ENDOFFILE

End of device input data detected by *read* command.

FUNCTION

Function got one or more unsupported argument values.

ILLEGAL VALUE

Math operation got result outside supported by CPU.

System variables $X and $Y got illegal values.

Date format is unsupported, function need $horolog date format.

Function $zdateh got unsupported date argument value, nonexisting date, for example February 30.

Function $ztimeh got unsupported time argument value, nonexisting time, for example 28 hours.

Function $zdate got argument outside supported range.

INTERRUPT

Process detected Ctrl+C is pressed in console or telnet mode.

INVALID BIT STRING

One of $bitXXX function detected argument has invalid or corrupted bitstring format.

JOB

New process failed to start.

The *kill* command cannot remove self process from ^$JOB.

Process calls to nonexisting job entry in ^$JOB.

The job command got in new process environment on only local variables.

Data transfer to new jobbed process failed.

The *job* command cannot accept local variables by reference.

The *job* command got invalid routine name.

The *job* command cannot insert new record into jobs list, job list is full. Job count limit is specified in minim.ini configuration file and in licence file.

Process list structure corrupted, *job* command failed to allocate one more entry.

LABELREDEF

Failed to compile routine with doubled label names. Two or more labels with the same name in one routine are not supported.

LIBRARY LOAD

Failed to find external ZDLL module file, or this file is not dynamic load library, or this library does not contain ZDLL initialization, or ZDLL initialization failed on load.

LIST

One of $listXXX function failed to use argument as a list structure, list format corrupted.

MAXNUMBER

Number overflow in math operation, result does not supported by CPU.

MAXOBJCODE

Error while compile routine to bytecode, result exceed bytecode limit (32Kb).

MAXSTRING

String operation failed to create unsupported string length (32Kb).

MEMORY

Operating system does not grant enough memory value.

MINNUMBER

Number overflow in math operation, result does not supported by CPU.

MNEMSPACE

Unsupported routine name to handle mnemonics.

NAKED

Naked indicator is an empty string.

Naked indicator have not any subscripts.

NAME

Syntax error, unsupported local, global or structured system variable.

NAMESPACE

Call to nonexiting database using extended syntax for routine or global variable.

NOLINE

Specified label or label + offset does not found to execute by *do* or *goto* commands or user function.

NOROUTINE

Call to nonexisting routine to execute.

NOTOPEN

Specified device does not opened.

NOZROUTINE

Process cannot find routine to handle z-function.

NULL VALUE

List function detected list element has undefined value.

PARAMETER

Label has arguments but called without parenthesis.

Label has not arguments but called with arguments.

Label accept less arguments then passed.

PARAMLIMIT

Syntax error, passed more than supported arguments (maximum 63).

PATTERNLIMIT

Pattern complexity exceeds MiniM possibility, it is required much more memory to handle pattern then MiniM supported.

PCRE

PCRE functions got syntax-incorrect regular expression.

RANGE

Some of $listXXX function detected list range specified incorrecly.

Some of $bitXXX function detected bit position specified incorrectly, must be between 1 and 262104 inclusively.

Convertion function $zcvt detected incorrect input format or data length.

READ

Read commands not applicable to device in current mode, read data not allowed or not supported.

Data transfer on *read* command failed at hardware or driver level.

SELECT

No any $select cases are successful.

SELECTARGS

Function $select does not support so much arguments (maximum 255).

STORE

Local process storage has not enough memory block to store local variable.

SSVN VALUE

Read of this structured system variable not supported.

SSVNSUBSCRIPT

This structured system variable does not support one of specified subscript.

This structured system variable does not support so much subscripts.

STACKLIMIT

Process cannot create one more stack level, stack count exceed specified in minim.ini configuration, section Process, key FrameCount. By default used 1024.

STRINGSTACK

Internal memory to calculate and pass temporary values fullfilled.

SUBSCRIPT

Empty string as subscript does not allowed.

Subscripts count exceed maximum count (63).

Total name length exceed maximum length (255).

SYNTAX

Syntax error - illegal spaces, unsupported command, system function, variable name, operator, or unsupported punctuation.

Illegal command, $text, name or other indirection expression value.

SYNTAX INDEX LIMIT

Syntax error, subscript limit exceeded (maximum 63).

TLEVEL

Command exceed maximum allowed transaction level count.

UNDEFINED

Read of undefined local or global variable name.

The *for* command variable got undefined value.

UNIMPLEMENTED

This command, command argument, function, system variable or structured system variable not implemented in current MiniM version.

WRITE

Failed to write any data to device not allowed or not supported for this device mode.

The *write* command failed to transfer data at hardware or driver level.

13.3 MiniM system errors list

Here are described MiniM system errors which info logged into minim.log file. Some system errors may be fixed by server administrator, and to fis other errors need to contact MiniM Database Server technical support at support@minimdb.com.

Error # 0. MiniM cannot find errors file description. This file from MiniM installation directory must have the minimerr.ini name and must be in the /bin subdirectory. If your MiniM installation contains localized files, for example, minimerr.dan.ini, you can copy this file to the minimerr.ini file.

Error # 1. MiniM cannot find current installation name. In need to be present in configuration file minim.ini, section Server, key InstallName and must be test string up to 31 symbol. For example:

```
InstallName = MINIM00
```

Error # 2. Failed to create common server synchronization area. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 3. Failed to open minim.log file to write. Check MiniM Database Server accounts have enough permissions. Loggin failure does not crash any data, but can crash MiniM process.

Error # 4. Failed to delete prior minim.log.bak file. Check server processes have enough permissions to delete this file.

Error # 5. Failed to rename file minim.log into minim.log.bak. Check server processes have enough permissions.

Error # 6. Failed to write to minim.log file. Check server processes have enough permissions.

Error # 7. Failed to get temporary process memory. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 8. Failed to load national collation file %s.N, or this file corrupted. This file must be present in the /nat subdirectory and be a MiniM national collation file.

Error # 9. Failed to open shared internal server area. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 10. Common memory access error. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 11. Common memory areas creation error. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error #12. Internal synchronization objects creation failed. Normal MiniM Database Server functioning is impossible. One of the resent causing may be operating system resources exhaust and may be fixed to review concurrent work MiniM Database Server with some of currently running applications.

Error # 13. Failed to create common server locking area. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 14. Failed to access common server lock area. It is critical error, notmal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error #15. Failed to create common server configuration area. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 16. Failed to access common server configuration area. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 17. Failed to create common server configuration area. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 18. Server configuration file does not contain any database defined. Without defined databases MiniM database server cannot work properly. Need to be described, at least, *%sys* and *temp* databases. Initial typical MiniM installation contains *%sys*, *temp*, and *user* databases.

Error # 19. One of defined database does not contain root database file. Root database file is mandatory for MiniM database.

Error # 20. Failed to determine database extent specification. Extent data file name must be start with "extent" with following extent number, for example:

```
extent1 = disk:/filename1.dat
extent2 = disk:/filename2.dat
extent3 = disk:/filename3.dat
```

Error # 21. Failed to determine one of the extent number specification. Extent data file name must be start with "extent" with following extent number:

```
extentNNN = filename,
```

here NNN must ne decimal extent number. Extents must follows in database configuration one-by-one.

Error # 22. MiniM database configuration file minimdb.ini contains unsupported key. Check minimdb.ini file sections and keys names.

Error # 23. One of database name configured exceed maximum database name.

Error # 24. One of data file in database configuration file minimdb.ini exceed maximum possible file name length.

Error # 25. One of configured MiniM database name contain forbidden symbols.

Error # 26. Failed to create internal server page lock area. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 27. Failed to create internal server page caching area. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 28. Failed to create internal server page cache descriptors area. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 29. Failed to create internal server cache queue area. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 30. Failed to create page cache synchronization objects. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 31. Failed to create page lock semaphors area. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 32. Failed to create page cahe semaphore object. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 33. Failed to create internal server statistic's area. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 34. Failed to use internal server statistic;s objects. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 35. Failed to create write daemon synchronization object. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 36. Failed to create journal daemon synchronization object. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 37. Failed to create internal server journaling queue area. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.

Error # 38. Failed to read journal file header. It is a critical error. Check file access permissions are enough, restore database from backup or remove journal and backup all databases.

Error # 39. Failed to write journal file header. It is a critical error. Check file access permissions are enough, restore database from backup or remove journal and backup all databases.

Error # 40. Failed to write to current journal file. Check write permissions for journal daemon (minimjd.exe) are enough and server has free disk space.

Error # 41. Failed to open before image journal file minim.bij. Check write daemon (minimwd.exe) have enough permissions and server have enougth free disk space.

Error # 42. Failed to write to before image journal file minim.bij. Check write daemon (minimwd.exe) have enough permissions and server have enougth free disk space.

Error # 43. Failed to create shared routine cach area. It is critical error, normal MiniM Database Server functioning is impossible. The most frequent causes may be hardware failure and insufficient virtual memory available.