Description of MSH language

Content

Оглавление

1.Introduction

Intention

Intention is to reduce servicing time of the software life cycle.
Reduction of the time for the development is achieved by following means:
-Language should be syntactically simple, neat and as low as practicable
-Language structures should be unambiguous and should not enable unusual usage.
-Language instruction set should be sufficient to support all current models of programming.

-Language should be flexible to the possible extent.
This is achieved by absence of the declared variables.
Reduction of the debugging time is achieved by following means:
-Simplicity and visibility of the programs.
-Minimum possible errors in the program.
-Programs should include minimum strings.
Reduction of the maintenance time is achieved by the following means:
-Language reliability.
-Language should provide supporting means for the data integrity.
- Minimum possible errors in the program.

None of the current language conforms to the stated objective. There is wide selection of various languages available, each of which carries some ideas. But there are not that many ideas. I will try to use the most valuable ones in this language.
In order to achieve maximum flexibility language should offer certain properties, in particular language should not be declared. All declared languages lose their flexibility promptly. Programs based on those languages, promptly swell. Declarations of the variables themselves apart from helping the translator, do not have any other use. Language should intend to manipulate the data and it should not contain user interaction dialog building tools.
MUMPS is the programming language that most closely corresponds to the desired goal. We will use it as the basis. The latest standard of this language is the standard of the 1995, this will be a starting point.

2.Language vocabulary

Alphabet
It is assumed that the Alphabet is the set of symbols in UTF-8 code.

Comments
Single-line comment //
Multiline comment /* */
When describing the language optional parameters are placed inside the brackets <>

3.Operations

Any variable could be operands: constants and functions
Unary operations:
'-' Unary minus is the feature property of the negative number.
Result: Arithmetic type.
'~' Logical operator NOT
Result : integer 0 or 1
Binary operators:
Operations can be 1-2 character.
Arithmetic operations: Result Arithmetical type
'+' plus
'-' minus
'*' multiply
'**' power
'/' division
'\' exact division Result -integer
'#' module of the number Result integer
Logical operations : Result integer 0 or 1
'&' task AND

'|' task OR
Comparison operators: Result integer 0 or 1
'>' greater-than
'<' below
'==' equal
'~=' not equal
'~>' not greater than
'~<' not less than
'>=' greater or equal
'<='less than or equal
'<>' not equal
String operations: Result- character string
'_' string concatenation
'_>' string follows
'_<' string contains
Bitwise operations on strings:
'_ &' AND operation
'_ |' OR operation
'_ ^' XOR
'_ ~' NOT operation
Selecting operation:
'?'-complies with *C* language standard nearly.
ExpYcl ? ExpTrue ! expFalse
expYcl- selection condition,
expTrue- expression assigned when conditions are executed
expFalse- expression assigned when conditions are NOT executed .

Built-in operations have the form of transactions with the prefix `
Arg1 `name Arg2

4.Constants

There are no constants in MUMPS. In MSH they are entered. But because of the specific nature of the language, they cannot exist as unchangeable variables of the program execution time. Therefore, in MSH they are entered as compile-time objects. Something like the define C language in the most simplified form. At the moment of translation of the MSH language into Pi, the virtual machine code, all constants in the source text are replaced with their values. That is, they are no longer in the Pi code. Constants can be used as variable values, as an index or variable name.
Constants consist of name and value. Name is an identifier. Constant value
There are any number of alphanumeric characters in any languages. The constant value can be in quotes or in double quotes. In this case, any characters may appear inside the quotes.
Command format:
Constant *nameConst = valConst <, nameConst = valConst>*;

To the right of the equal sign can only be a constant, the expression is not allowed. The name of a constant must be unique and do not overlap with the names of variables.
Example of constant values:
Constant cCnst1 = 123 ABC, cCnst2 = 15Ngsh, cCnst3 = '75 & 56 + "()" ', cCnst4 = "wq ^ erNG'";
The second form of the constant is:
Constant nameMod <, nameMod>;
nameMod is the name of the module from which the constants will be imported. This will allow you to have libraries of constants.

Example:
Constant modLibSys, modLibUser;
Where modLibSys and modLibUser are modules containing constants. At the time of broadcast, they must exist.

5.Variables

In MUMPS variables are stored in a wooden structure, whether it is globalization or locale. The truth at the global in MUMPS there is a small difference - the abbreviated syntax. The MSH is not abbreviated syntax. Storing data in the form of only the tree allows you to standardize data access. You can write the treatment program of any tree data. But in the programs often have to work with a variety of intermediate values and store them in the tree is not effective. Access to the tree always entails additional expenses, as we would not have optimized this access. In MSH data may be stored in a wooden frame and a one-dimensional array. In the one-dimensional array is better to store intermediate data, although it is possible to store any data. Access to the array much faster than a tree. Wood kept the so-called sparse data. That is, in the tree stores only the existing node. For example, we have a tree [1] [5] [78]. Only the top and it will be stored in the tree. An array stores all intermediate values. For example, we created a 78 element array in $ 78 all items will be created in the array until 78.

Visibility variables in MUMPS has its own characteristics. The fact that called MUMPS globals in other languages is a reference to databases. But locale are analogous to variables in other languages. Well, everything is clear with the global. This external data, and they are available as of any assignment of this application and from other applications. They are stored as files, and is usually more global in a single file. In MSH generally all the same except for one point. Each global is stored in a separate file. And to be precise it in multiple files one directory. This point is significant from a technological point of view, the construction of an information system. And one more remark. Global in MSH synchronized handling. This means that they can be accessed simultaneously by multiple tasks without any additional synchronization.
Now let's deal with more detailed locales. In general, the locale in MUMPS is a global variable in terms of top-level C language. Analog variables from the heap in C, in MUMPS not. And here is an analogue automatic variable Xi has a language. This locale listed (listed or not), depending on the shape of the team, the team New. The scope of such locales on the New command to end the command unit Quit. The original course due to the nature of the decision and the lack of MUMPS in it variable declarations. In my opinion, the global scope by default locale controversial decision, but not fatal. In MUMPS where I mostly work directly with global, local variables are not so many names and conflicts are not frequent, although happen. But still it is not convenient for programming. The MSH adopted a different approach. Teams do not have New here. A localization variables is made by prefix. Inside the unit Do localized only one variable without a name. The variables with the prefix % localized within the application. Variables do not have these prefixes are located inside the unit Job. This applies equally both trees and arrays.
In language there are no variable declarations. Partly it arises at the moment when it is set. When referring to an undefined variable returns an empty string.
The variables are the external memory is called globalization and are prefixed with ^.
For example: ^ abc [15, gh8,42] abc- the name of globalization.
The variables are in memory are called locale.

Locales are divided by scope.
1. Variable block Do. This variable is visible only within the block Do. This variable has no name. For example [1,2] or $ 4.
2. Application variables. They are common to all application programs. These variables have the prefix %

For example %a1[1,2] or %a2$4.
3. The rest of the variables. They are visible only within a given job.
Variables can have two storage structures.
For example a1[1,2] or a2$4.

The tree structure.
Such variables appear in the form of an arbitrary index structure in square brackets. Indices separated by commas. Any index can be of any type. The number, string, etc.
For example: ldb [abc, 125,5,9]
Indices variables may be arbitrary expressions.
The predefined constant name %this can be used. In this case there is an appeal to the private properties of an object. This treatment is only possible within the method or property of an object.
For example:
[%this] .Age

The structure of a solid array.
This structure has one integer index. All the elements of the array are arranged in the row. This structure is used for quick access to data.
Array index follows the name$sign
For example: db$25
The array index begins at 1. The zero element of the array is a service, and the index of the last element. Iteration ignore this element.
The program will be available as a complete MSH referred to as the specific data of the tree or array, and an abbreviated reference to the tree or the entire array. The full link contains a variable name and a full index.
For example:
ab[a, gt, 125] or the ab$25.
shortened link contains a name and an empty index.
For example:
ab [] or the ab $
Constructions where permissible abbreviated reference must be agreed separately.
A link to a node is not provided within the tree. It is also possible to refer to a tree and an array at the same time, then only the name is indicated.
For example:
ab
Where possible, such an appeal is negotiated specifically in a specific team.

6. Pseudo arrays.

They service information is transmitted. Appeal to them the appropriate treatment to the array. They are open for reading and usually closed on entry.
A list of pseudo arrays:
prefix A$- the values passed to the function arguments, opened for writing,
prefix B$ - Input output buffer, opened for writing,
prefix J$ call stack of tasks for the current block Do, Open only on reading.

7. Expressions

Expressions are combinations of operands and operations. Operands can be expressions, when they are set in brackets. Types of variables are random. The result of the operations is uniquely defined by operation. There are no operations priorities.

8. Commands

Arguments in different commands can vary. Argument can consist of subfields, that are divided either by symbol =

For the further explanations let us assume following conventions:

exp — prefix that indicates an expression

expYcl — expression that sets conditions for command execution

ref — prefix that indicates reference to the variable.

refTree - a reference to the variable tree structure.

Prog - Reference to the program in the form of *<expMod.>expLabel<(expArg<,expArg>)>*

expMod — module name can be an expression

expLabel —label in a module can be an expression

expArg - Argument can be an expression

Time intervals are set in microseconds 10e-6 sec

Declarative commands:

Parent , Constant.

Declarative commands have following format:

CMD *Arg <,Arg>;*

CMD — command code.

Arg — command argument.

Declarative commands are executed at the moment of translation as they appear in source text of the program. Execution of those does not depend on progress of the program and does not contain conditions of their execution. The condition of the command does not apply to them.

Parent command.

Arguments of the command are Parents of this module. Command is used to organize inheritance.

Parent *expParent <,expParent>;*

expParent- constant name of this module. As it is clear from syntax, multiple inheritances take place. Parent selection priority is defined by the sequence in the list of inheritance. The sooner Parent appears the higher is it's the priority.

Constant (Const) command.

In MUMPS constant no. The MSH are introduced. But because of the specificity of the language exist as immutable variable execution time of the program, they can not. Therefore, MSH are introduced as objects of compile-time. With something like define the C language in its simplest form. At the time of MSH language broadcast Pi code for a virtual machine, all the constants in the source text are replaced by their values. That is, in the Pi code, they're gone. Constants may be used as the values of the variables as an index or a variable name.

To the right of the equal sign can only be a constant expression is not allowed. The name must be unique constant and variable names do not overlap.

Constant command has 2 forms.

1 form:

Constant *nameConst=valConst <,nameConst=valConst>;*

nameConst- name of the constant should be an identifier and name length should not exceed 18 byte,

valConst- value of the constant.

2 form:

Constant *nameMod <, nameMod>;*

nameMod - module name which from to import the constants section.

Execution commands:
Break, Back, Back1, Copy, Run, Do, Else, End, Case, While, GoTo, If, Job, KiLL, LockW, LockR, LockUn, Move, Next, Next1, Query, Query1, Return, Set, TimeOut,Transaction,Try, EventTrap, EventCall, EventWait, EventDelete, Write, Read.

Commands consist of :
-the command code
- nonrequired execution condition
-required space and
-nonrequired arguments that are listed and separated by comma.
End of the command is marked with semicolon. Command code does not depend on register and can be reduced. Full command code can be expanded by any amount of Roman characters or numbers. E.g command *End* could be recorded as *EndIf*. Same applies to other commands.
Command execution condition is separated from command code by character*?*
If the result of the execution condition evaluation is not equal to 0 then the command is executed.
CMD*<?expYcl> <Arg1 <,ArgN>>;*

Expression evaluation command **Set**.

Set (S) - appears like:
Set*<?expYcl> ref=exp <,ref=exp>;*
S*<?expYcl> ref=exp <,ref=exp>;*
The variable *ref* is assigned the value of the *exp* expression.
For example: **Set** ab [1,2] = $ 1 + 78 * 7;
The shortcut **Set** does not have a right side
Set<?*expYcl*> *exp*<,*exp*>;
S<?*expYcl*> *exp*<,*exp*>;
Inside the exp expression, there must be a reference to variables, the last of which is assigned the value of the *exp* expression.
For example: **Set** 9- [2] -ab [1,2] + 78 * 7;
Variable ab [1,2] will be assigned the value of the expression.
For example: **Set** $1=10; **Set** $1+2;
Variable $1 will be assigned the value of 12.

Data deletion command **Kill.**

Command **KiLL (K).**
Kill*<?expYcl> ref<,ref>;*
K*<?expYcl> ref<,ref>;*
Command deletes the descendants of the node and the node itself ref and object properties. This command can be used a shortened link.
There are more modifications of this command.
Deletes data in the *ref* node .
KillD*<?expYcl> ref<,ref>;* **KD***<?expYcl> ref<,ref>;*
Deletes descendants of the *ref* node
KillN*<?expYcl> ref<,ref>;* **KN***<?expYcl> ref<,ref>;*
Removes the object properties node ref.
KillP*<?expYcl> ref<,ref>;* **KP***<?expYcl> ref<,ref>;*
Combinations such as postfix
Deletes data and properties of the node *ref*.
KillDP*<?expYcl> ref<,ref>;* **KP***<?expYcl> ref<,ref>;*

Deletes data and descendants of the node *ref*.
KillDN<*?expYcl> ref<,ref>;* **KN**<*?expYcl> ref<,ref>;*
If you want to delete only the tree, then the command is used.
Kill ^gb[]
And if only the array is
Kill ^gb$
In this command it is also possible to specify only the name. In this case, both the tree and the array with this name will be deleted.
For example:
Kill ^gb
Obviously, such a command cannot be used for local variables of the Do block. Due to the lack of a name for such a variable.

Data copy command **Copy** , **Move**.

Data node *ref From* merges with the *refTo* node. Shortened links can be used in this command.
Copy<*?expYcl> refTo=refFrom<,refTo=refFrom>;*
Data node to node moves *refFrom refTo*
Move<*?expYcl> refTo=refFrom<,refTo=refFrom>;*

Task locking commands **Lock**.

LockW (LW).
LockW<*?expYcl> expID<,expID>;*
LW<*?expYcl> expID<,expID>;*
It sets locking on identifier assigned by *expID* expression by writing

LockR (LR).
LockR<*?expYcl> expID<,expID>;*
LR<*?expYcl> expID<,expID>;*
It sets locking on identifier assigned by *expID* expression by reading

LockUn (LU).
LockUn<*?expYcl> expID<,expID>;*
LUn<*?expYcl> expID<,expID>;*
Cancels locking of identifier with assigned *expID* expression.

1 Program progress change command.

TimeOut command.

Holds-up execution of the current program.
TimeOut<*?expYcl> expTime<,expTime>;*
expTime **—** expression that sets evaluation delay in the current program.
Time intervals are set in microseconds 10e-6 sec

GOTO (G) command.

GOTO (G) command is the transition to the new program execution point.
GOTO<*?expYcl> expMod.expLabel< (arg <,arg>) >*;
G<*?expYcl> expMod.expLabel< (arg <,arg>) >;*
Transition to the tag in same module.
GOTO<*?expYcl> expLabel< (arg <,arg>) >;*
G<*?expYcl> expLabel< (arg <,arg>) >;*
expMod **—** the evaluation result of this expression is the name of the module.

expLabel **—** the evaluation result of this expression is the continuation point of the program execution.
When you switch to the **GOTO** command, you can replace the arguments with those specified in parentheses.

Do(D) command.

The team performs a subroutine call without variables localization. Inside the block, only a variable with no name is localized.

Do*<?expYcl> <expMod.>expLabel< (arg <,arg>) >< ,<expMod.>expLabel<(arg<,arg>)>> ;*

Job(J) command.

The command executes the program in the new job. For each such call, a new block Do. Upon completion of the assignment generated code completion. During the job status code is available job.

Job*<?expYcl> <expMod.>expLabel<(arg <,arg>)> <,<expMod.>expLabel<(arg <,arg>) >> ;*

Return (Re) command.

This command completes execution of the running program and returns back the value to the calling program.
Return*<?expYcl> expRet;*
Re*<?expYcl> expRet;*
expRet- expression is evaluated and returned to the calling program.

Break (Br) command.

Exit from the loop command can be executed in any point of the loop by using command ***Break(Br)***.
If the **Break** command is encountered within the **Case** or **If** or **Else** command, then the output occurs beyond all the attachments of these commands.
Break<?*expYclFor*>;
Br<?*expYclFor*>;

Xecute (X) command.

The command evaluates the argument, interprets it as an MSH command, and executes it.
Xecute <? ExpYcl> exp <, exp>;
The exp expression is evaluated, interpreted as an MSH command, and executed.

2 Block command.

A program may contain blocks. The block consists of 3 parts.
1. The title of the block, which is a block command. The block command determines the type of block.
2. The body of the block, consisting of any number of teams.
3. The end of the block, the **END** command.

Block command **If (I)** and **Else(E)**.

If condition of execution of this program is not equal to 0, then commands that follow *If* command up to *Else* or *End* command are executed. Inside *IfEnd* block *Else* commands may take place.

Else command without execution condition can be only single one and should be the last one in a

row of *Else* commands. If *Else* command has execution condition , and this condition is not equal to 0, then commands that follow after it up to *Else* and *End* command are executed. These commands do not have arguments.

If?*expYcl*;
I?*expYcl*;
Any commands
Else<*?expYcl;*>
E<*?expYcl*>*;*
Any commands
END;
Else command in the format with execution condition is practically the ***ElseIf*** command, without execution condition is the command ***Else***.
In the overall view block **If** could look like this :
If?*expYcl*;
comand...
<**Else**?*expYcl*;
comand...>
<**Else**;
comand...>
End;

Selection block command **Case**.

The command ensures the execution of a section of code depending on the specified condition *expYcl*. The condition in the command is mandatory. The command has no arguments and is used in Pascal notation.
The result of evaluating the expression *expYcl* should give a string.
If this line is equal to one of the labels inside the **Case** block, then the commands following this label are executed to the next label or **End** command.
If the string is not equal to any label then the commands <*comandElse* ...> are executed. Case commands can be nested as well as combined with any commands.

Case?*ExpYcl*
<*comandElse* ...>
Label: *comand* ...
<Label: *comand* ...>
End;

For example:
Set $1 = L1;
Case?$1
Set $2 = ’label did not match’;
L0: **Set** $2 = ’label L0’;
Set $3 = L0;
L1: **Set** $2 = ’label L1’;
Set $3 = L1;
L2: **Set** $2 = ’label L2’;
Set $ 3 = L2;
End;

Block command **Transaction(Tr)**.

Transaction command is the beginning of the transaction execution block of the globals. Action of this command applies only to the current job.

In case if inside this block *Job* command occurs, then the current transaction is not applied to the *Job* execution. Block ends with *End* command with one argument. If this argument has value w=0, then transaction rollback is performed. If Argument is missing, then transaction is conducted. If the command execution condition is false, then the whole block is not processed.
Transaction<*?expYcl*>;
comand...
End <*exp*>;

While block commands.

Command does not contain loops initializations. Loops initialization is executed with help of other commands before loop. If execution condition of **While** commands becomes equal to 0, then loop is terminated. If execution condition of **End** commands becomes not equal to 0, then loop is terminated.

While<*?expYclWhile*>;
comand...
End<?*expYclEnd*>;

expYclWhile **—** loop continuation condition
expYclEnd **—** loop ending condition

3 Block loop iterators.

These commands are used for data traversing. On each step following node with data becomes accessible.
* Note on the implementation teams for the **Next, Back,** and **Query**. These remarks do not apply to teams **Next1, Back1** and **Query1**:
Inside the iterator block reference index should not change. Expressions such $$2 or [[3]] unacceptable if $2 or [3] are changed inside the iterator. Changes in these variables will not be taken into account.
During traversal data structure should not be changed.

Next (N) command.

Traversing the data tree.

Node descendants' traversal command on one level from the end to the beginning. It can be used a shortened link.
Next<*?expYcl*> *refY<,refSaveInd>;*
comand...
End<*?expYclEnd*>;
N<*?expYcl*> *refY<,refSaveInd>;*
comand...
End<*?expYclEnd*>**;**

r*efY* — reference to the node, which descendants will be traversed. The reference index. It can be used a shortened link.
refSaveInd - reference to the node, where sub index of the descendants will be saved. Only latest index of the descendant will be saved. The parameter is optional.
Node data available in the system variable %queryData. The node index is available in the %queryKey variable. If you want to bypass the node properties, the index is added to the symbol ':' If the descendants need to be bypassed not from the beginning, then the index after which it is necessary to begin the walk is written to the refSaveInd variable, otherwise this variable must be

deleted before the walk command **Kill**.
<*?expYcl*> - condition to continue the cycle.
<*?expYclEnd*> - the condition under which the cycle will end. If the condition is not 0, then the cycle will end.

For example **Next** [],$1
For example **Next** ^g[],$1
For example **Next** b1[4:],$1
This link should not be changed inside the iterator. Any change will not be taken into account.
For example **Next** aa[4], $$1
In this case, only the first will be used the value of $1

Array traversal.

The team bypasses all array elements in a row. Without missing empty values.

Next<*?expYcl*> nameMacc <, refSaveInd>;
comand ...
End<*?expYclEnd*>;
N<*?expYcl*> nameMacc <, refSaveInd>;
comand ...
End<*?expYclEnd*>;

For example **Next** $, $1
For example **Next** ^g$, $1
nameMacc is the name of the array. Only an abbreviated reference should be used.
refSaveInd - link to the node in which the child subindex will be saved. Only the last child index will be saved. Parameter is optional. Node data is available in the% queryData system variable. The node index is available in the variable% queryKey. If the descendants need to be bypassed not from the beginning, then the index after which it is necessary to start the crawling is written to the refSaveInd variable, otherwise this variable must be deleted with the **Kill** command before crawling.
<*?expYcl*> - condition to continue the cycle.
<*?expYclEnd*> - the condition under which the cycle will end. If the condition is not 0, then the cycle will end.
This point relates to the same teams **Back** and **Query**.

Back(Ba) command.

Traversing the data tree.

Node descendants' traversal command on one level from the end to the beginning. It can be used a shortened link.
Back<*?expYcl> refY<,refSaveInd>;*
comand...
End<*?expYclEnd*>;
Ba<*?expYcl> refY<,refSaveInd>;*
comand...
End<*?expYclEnd*>**;**

r*efY* — reference to the node, which descendants will be traversed. The reference index. It can be used a shortened link.
refSaveInd - reference to the node, where sub index of the descendants will be saved. Only latest index of the descendant will be saved. The parameter is optional.
Node data available in the system variable %queryData. The node index is available in the

%queryKey variable. If you want to bypass the node properties, the index is added to the symbol ':' If the descendants need to be bypassed not from the beginning, then the index after which it is necessary to begin the walk is written to the refSaveInd variable, otherwise this variable must be deleted before the walk command **Kill**.
<?expYcl> - condition to continue the cycle.
<?expYclEnd> - the condition under which the cycle will end. If the condition is not 0, then the cycle will end.

For example **Back** [],$1
For example **Back** ^g[],$1
For example **Ba** b1[4:],$1
This link should not be changed inside the iterator. Any change will not be taken into account.
For example **Back** aa[4], $$1
In this case, only the first will be used the value of $1

Array traversal.

Back*<?expYcl>* nameMacc <, refSaveInd>;
comand ...
End*<?expYclEnd>*;
Ba*<?expYcl>* nameMacc <, refSaveInd>;
comand ...
End*<?expYclEnd>*;

For example **Back** $, $1
For example **Back** ^g$, $1
nameMacc is the name of the array. Only an abbreviated reference should be used.
refSaveInd - link to the node in which the child subindex will be saved. Only the last child index will be saved. Parameter is optional. Node data is available in the% queryData system variable. The node index is available in the variable% queryKey. If the descendants need to be bypassed not from the beginning, then the index after which it is necessary to start the crawling is written to the refSaveInd variable, otherwise this variable must be deleted with the **Kill** command before crawling.

<?expYcl> - condition to continue the cycle.
<?expYclEnd> - the condition under which the cycle will end. If the condition is not 0, then the cycle will end.

Query (Q) command.

Traversal command of all descendants of the node *Query.*
Query command traverses tree branch from top to bottom and from left to right
Query*<?expYcl> refY<,refSaveInd>*;
comand...
End*<?expYclEnd>*;
Q*<?expYcl> refY<,refSaveInd>;*
comand...
End*<?expYclEnd>***;**

refY — reference to the node, which descendants will be traversed. The reference index. It can be used a shortened link.
refSaveInd - reference to the node, where sub index of the descendants will be saved. Saved the entire index in a list. The parameter is optional. If the descendants need to be bypassed not from the beginning, then the index after which it is necessary to begin the walk is written to the refSaveInd

variable, otherwise this variable must be deleted before the walk command **Kill**. Node data available in the system variable %queryData. The node index is available in the variable %queryKey.
<?expYcl> - condition to continue the cycle.
<?expYclEnd> - the condition under which the cycle will end. If the condition is not 0, then the cycle will end.

Array traversal

The command is similar to the Next command. But empty values are skipped.

4 Do not block the team Teams traversal tree.

Next1 (N1) command.

Next1 (N1) command provides next node on the same level
Next1*<?expYclEnd> refY,refSaveInd;*

refY — reference to the node, which descendants will be traversed. The reference index.
refSaveInd - reference to the node, where sub index of the descendants will be saved. Only latest index of the descendant will be saved. This parameter is required. Node data available in the system variable %queryData.

Back1 (Ba1) command.

Back1 (Ba1) command provides previous node on the same level .
Back1*<?expYclEnd> refY,refSaveInd*;

refY — reference to the node, which descendants will be traversed. It can be used a shortened link.
refSaveInd - reference to the node, where sub index of the descendants will be saved. Only latest index of the descendant will be saved. This parameter is required. Node data available in the system variable %queryData.

Query1 (Q1) command.

Query1 (Q1) command provides next tree branch during traversal of the whole node from top to bottom and from left to right.
Query1*<?expYclEnd> refY,refSaveInd*;

refY — reference to the node, which descendants will be traversed. The reference index. It can be used a shortened link.
refSaveInd - reference to the node, where sub index of the descendants will be saved. Only latest index of the descendant will be saved. This parameter is required. Node data available in the system variable %queryData.

5 Event Processing Commands.

Command **EventTrap (EvT)**.

This command creates a custom event.
EventTrap*<?expYcl> expName < ,expName >*
EvT*<?expYcl> expName<(arg1<,argN>)><,expName<(arg1<,argN>)>>*
expName — Expression that sets name of the event. Event name cannot exceed 18 bytes
arg - arguments that will be transmitted to the handlers of this events.

By this command all handlers of this event will be initialized and arguments will be transmitted to them. Streams waiting for this event to occur will be continued. From this moment event arguments will be available for them.
On the team will be running in separate threads all handlers of this event and they will be passed arguments.
Upon completion of all event handlers, it will be deleted.
If no handler for this event that will wait for the event handler will then be processed and removed.

Event processing commands

Command **EventCall (EvC)**.

Command assigns handler for an event ending.
EventCall<*?expYcl> expName=Prog<,expName=Prog>*
EvC<*?expYcl> expName=Prog <,expName=Prog >*
expName — expression that sets event name
Prog —reference to the program-event handler
If at the moment of the execution of this command event already exists then handler will be called and the event will be deleted.

Command **EventWait (EvW)**.

Command -Wait for event termination.
EventWait<*?expYcl> expName <,expName >*
EvW<*?expYcl> expName <,expName >*
Waiting for the event to start and when it starts then task execution is resumed. From this moment arguments that have been transmitted by the **EventTrap** command are available in the program. If at the moment of running of **EventWait** command event already exists then execution will be continued.
expName — expression that sets event name

Command **EventDelete (EvD)**.

The delete command event.
EventDelete<*?expYcl> expName<,expName>*
EvD<*?expYcl> expName<,expName>*
Clears events with the specified name.

Files processing command.

Input and output file through variable B$.
Write to the file command:
Write<*?expYcl> pathFile<(arg1 <,argN>)>;*
pathFile —string expression whose value is the file name.
The file output variables from an array of B$. If a separator is set in the system variable %dmtIO , then input the output stream between the values displayed separator.

Read from file command:
Read<*?expYcl> pathFile;*
pathFile —string expression whose value is the file name.
The entered values will be placed in an array B$. If a separator is set in the system variable %dmtIO, then input the output stream between the values displayed separator.

The variable name *pathFile* file, you can specify the standard output of the input device.
- ST device STDOUT
- ER device STDERR

- SR TCP server socket device, if the language is made of standard treatment data server.

9. Event.

Another effective programming paradigm is the event. Usually programming languages do not contain funds event. They are usually delivered in the standard libraries. In the MUMPS process events in its infancy and is reduced to handling errors. The MSH events are included in the language. Events can be either system generated by the system. And user generated by the user using EventTrap team. The arguments for this command are passed to handlers of this event. EventCall command assigns an event handler program. At the time of the event the current job will be interrupted and the program processor in the new block will be executed Do. As if this place Do team met the challenge of the handler. This means that the local variables of the program is executed in the event handler will not be available.
EventWait command stops execution of the current programs and looks forward to the emergence of the event. When an event occurs, the current program continues to work, and from that moment the program is made available its arguments. The new unit Do not create so all local variables of this block available after Do kaomandy EventWait.
If the event occurred until EventCall and EventWait teams. That event will be processed first encountered EventCall or EventWait team.
After processing, the event is removed.
EventDelete command removes the event along with the handlers.

Events can be caused by processes occurring in the system. These are system events and they have *sys* prefix. Events can also be called by **EventTrap** command in the program. These are user events and they have *usr.* prefix

System events:

sysJobEnd — Job completion in the transmitted argument is the number of the completed Job. This event occurs at the moment of ending of any job and doe not require *EventTrap* command

Systems events of the work with external devices.

This events are initiated by the **EventTrap** command. But handlers will be called only when event will be completed. When events handlers are called in the 1st argument, device identifier will be returned and it will be positive integer. If error occurs then error code is returned in this argument and it will be negative integer. 2nd argument, if it does exist, depends on the event.

SysRead — Device reading. Data that have been read from the device are transmitted in 2nd argument.

File identifier is transmitted in the command *EventTrap* as an argument.

SysOpenReadClose — Event occurs in any of these events.

Name of the file is transmitted in the command *EventTrap* as an argument.

10. Lists.

Any variable can be the list. List consists of one or number of values. Values can be any type including lists.
List can be created by structure:
ref={exp<,exp>}
ref — link(reference) to the variable where list will be recorder.
exp — expression that becomes relevant element of the list. List can be returned by *Query* command

If the reference *ref* is the list, then in the Index it will evolve into relevant number of indices.
<ref>

Each element of the list will become relevant Index. In the reference common indices can intersperse with lists. For example variable can look like :
[Af,[B,2],125,$1,5.6]
If variables[B,2] and $1 are lists then they will evolve into indices in accordance with list size.

11. Functions.

Functions could be built-in or user-defined. In-built functions are defined by the language. User-defined are developed by Programmer.
Name of the built-in function is set as constant. Name of the built-in function has prefix %.
Actual parameters are passed to the function by value and name. If the parameter is passed by the name of the variable name is supplemented behind the ! symbol.

1 Built-in functions.

Search Label in an arbitrary module.

%findLabel(*expLabel*<,*expMod*>)
The *expMod* expression whose result is the name of the module.

expLabel - expression the result of which is the name of the desired label.

The function returns either 1- if such a label exists, or 0- if not.

For example:

Set $1=**%findLabel**(*Lab,Mod*)**;**

Blocking functions.

%sysLockR(*exp,time*) — Locking by reading with waiting time,
%sysLockW(*exp,time*) — Locking by writing with waiting time,
%sysLockSt(*exp*) — Locking status
exp – expression, which value is Locking Identifier
Expression result should not exceed 18 bytes.
If the length is longer then only first 18 bytes will be considered.

time-expression whose value sets TimeOut in microseconds (usleep), if it is not specified, then it is assumed = 1.
Return value *ref*= locking result
0- locking by the current Job
<0-locking not completed
>0 -№ of the job, that has set locking
>0 —№ of job, that which has previously set locking. Locking has not been completed.

Processing functions complete the task.

%sysJob Status(*exp*) - status of the job, the job is completed 0, 1 job runs.
%sysJobErr(*exp*) - code of the job is complete.
%sysJobValue(*exp*) - the return value.
exp-expression value is the ID of the job.

Content file directory.

File subdirectories contained in this directory.

%sysReadDir(*expFile*)
expFile-expression specifies the name of the directory
Result is stored into the array B$.
The files contained in this directory.
%sysReadDirFile(*expFile*)
expFile-expression specifies the name of the directory
Result is stored into the array B$.

Recursive file directory contents.

File subdirectories contained in this directory.

%sysReadDirR(*expFile*)
expFile-expression specifies the name of the directory
Result is stored into the array B$.

The files contained in this directory.

%sysReadDirFileR(*expFile*)
expFile-expression specifies the name of the directory
Result is stored into the array B$.

Create a file directory.

%sysCreateDir(*expFile)*
expFile-expression specifies the name of the directory

Delete file directory.

%sysDeleteDir(*expFile*)
expFile-expression specifies the name of the directory

Delete the file.

%sysDeleteFile(*expFile*)
expFile-expression specifies the file name

Functions return number of processed files.

Compile the file.

The function of compiling the file:
%**sysCompile***(nameMod)*;
nameMod is an expression whose string value is the name of the module msh. The module is searched in the standard directory for source programs ./msh
The translation will create a command file of the MSH virtual machine in the ./mvm directory.
The return code is a 32 bit vector:
component 0 - return code, either 0 or a negative number (error code),
component 1 is the line number where processing was completed,
component 2 is the position number where the processing was completed.
If return = 0, then the line number is the last line processed and the position.
For example, we are broadcasting the byxUsrUI module:
Set $1 =**%sysCompile**(byxUsrUI), $2 = $1.**%v32**(0), $3 = $1.**%v32**(1), $4 = $1.**%v32**(2);
$2 - contains the return code, $3 - the line number where the processing was completed, $4 - the

number of the position where the processing was completed.

2. User functions.

User functions are common subprograms and have **Do** command argument format.
<expMod.>expLabel<(expArg <,expArg>)>
expMod — expression that sets module name.
ExpLabel - expression that sets entry point(subprogram name)
expArg — expression that sets argument.

12. Predefined properties.

1 System properties.

System properties provide additional information about the system.
%err - code of the last operation.
%this — Object Index
%queryKey - key to the last access to the data in the iterator.
%queryData - The value of the last access to the data in the iterator.
%Data - the value of the last access to the data in the Get command.
% eof - at the end of the iterator becomes equal to 1, otherwise 0,
%statData - state of the variable to which it was last accessed.
%dmtIO - field separator in the input output operations, is set for each job.
%nameMod - the name of the executable.
%idDiv — current input-output device
%isTr- transaction status. If the code is running inside transaction.
%idJob — ID of the running job.
%idDo — Block ID Do.
%statDo —Program execution status.

2 Properties of any variables.

Each variable has predefined properties
refData.**%type** –type of variable
refData.**%byte** - variable size in bytes in string form
refData.**%size** - size of the variable in characters in string form
refData.**%stat** - condition variable.
Result type is defined in accordance with property type

3 Data Methods.

Object constructor.

refData.**%objNew**(Type)

refData-reference to the tree data.

Type -type variable.

For example:

Do [1,2].**%objNew(nameObj);**

Set [1,2].**%objNew=nameObj;**

Data Bypass.

refData.**%sysQuery***(expLabel <, expMod, isProp>)*

expLabel - vertex processing function label,

expMod is the vertex processing function module.

IsProp - will the method bypass the properties of the node, if ~ = 0, then yes, when traversing the node, the properties of the node will be bypassed first, and only then its descendants.

The call to the processing program will be carried out in the new context and in the task of the current program.

This function is passed as arguments:

1. sign of the property, if this argument = 1, then the property of the node is transferred, otherwise the node is transferred,
2. the last node index
3. data in the node
4..N full path to the node.

The processing program must return 0 to continue.

Read ini file.

Running as a method of indexing data. pathFile - an expression whose value is a string filename. Reading takes place in the indexing variable refData.

refData.**%iniRead(***pathFile<,pathFile...>* **)**

For example:

. Do [1].**%iniRead («srv.ini»)**;

Write ini files.

Running as a method of indexing data. pathFile - an expression whose value is a string filename. Recording takes place in pathFile file.

refData.**%iniWrite(pathFile)**

For example:

. Do [1].**%iniWrite(«srv.ini»)**;

String properties.

Before these properties are executed variable is converted to the string format.

Operations on string character.

StartInd- index character, that indicated action should start with

Count — number of characters participating in actions.

1. Get *Count* string characters beginning with character with *startInd* index. If *Count* is not set, then it needs to be considered that it equals to 1:
   refData**.%strGet**(*startInd <,Count>*)
   Example [j1,j2]=[i1,i2,i3].**%strGet**(5,3)
2. Get string character code with index *startInd*
   refData.**%strCode**(*startInd*)
   Example [j1,j2]=[i1,i2,i3].**%strCode**(5)
3. Take characters from string node *Count*
   refData.**%strPop**(*Count*)
   example [j1,j2]=[i1,i2,i3].**%strPop**(2)
4. insert characters to the string beginning from character with *startInd* index

refData.%**strIns**(*startInd*)
Example [i1,i2,i3].***%strIns***(7)=[j1,j2]
or $2=[i1,i2,i3].**%strInsStr**(7,[j1,j2])

5. Replace characters in the *Count* string:
refData.%**strSet**(*startInd* <,*Count*>)
Example [i1,i2,i3].**%strSet**(4,3)=[j1,j2]

6. Delete from *Count* array characters beginning with character with Index *startInd*
refData.**%strDel**(*startInd* <,*Count*>)
Example:
[j1,j2]=[i1,i2,i3].**%strDel**(5,2)

7. Find substring beginning with character with Index *startInd* if *startInd* is not assigned then search should be started with start of the string
refData.**%strFind**(*subStr*<,*startInd*>)
Result: the desired position of the first character.
Example: [j1,j2]=[i1,i2,i3].**%strFind**(ABC,2)

8.In the string, the charSource characters should be replaced with the charRepl characters only 1 entry if charRepl is not specified, then the charSource characters should be removed from the string:
refData.**%strRepl**(*charSource*<,*charRepl*>)
Example:
[j1,j2]=[i1,i2,i3].**%strRepl** (ABC,123)

9. In the string, all charSource characters should be replaced with charRepl characters all occurrences if charRepl is not specified, then all charSource characters should be deleted from:
refData.**%strReplALL**(*charSource*<,*charRepl*>)
Example:
[j1,j2]=[i1,i2,i3].**% strReplALL** (ABC,123)

10. Replace characters in charSource charStr line on symbols charRepl.
refData.**%strUpDate**(charStr, charSource <, charRepl>)
Example:
[j1,j2]=[i1,i2,i3].**%strUpDate**(ABCDгщы,Bcг,bcГ)
Result: AbcDГщы

Actions on string field.

If *Count* is not set then it should be considered as equal to 1.

1. Get *Count* string fields starting with *startInd* field, if *Count* is not set then it should be considered that *Count*=1:
refData.**%fieldGet**(*delimiter,startInd* <,*Count*>)
Example
[j1,j2]=[i1,i2,i3].**%fieldGet**(',',5,3)

2. Add field to the string
refData.**%fieldPush**(*delimiter*)
Example
[i1,i2,i3].**%fieldPush**('.')=[j1,j2]

3. Take from string node of the *Count* fields if *Count* is not set then it should be considered that Count=1
refData.**%fieldPop**(*delimiter*<,*Count*>)
example
[j1,j2]=[i1,i2,i3].**%fieldPop**('.',2)
If the string to be added is not specified, then only a separator is added.
For example:
[i1, i2, i3].% fieldPush ('.')

4. Insert into the field string
refData.**%fieldIns**(delimiter,startInd]
Example
[i1,i2,i3].**%fieldIns**('/',7)=[j0,j1,j2]
5. Replace in the string *Count* fields beginning with *startInd* field
refData.**%fieldSet**(*delimiter,startInd ,Count*)
Example
[i1,i2,i3].**%fieldSet**('-',4,3)=[j1,j2]
6. Remove in string *Count* fields beginning with *startInd* field
refData.**%fieldDel**(*delimiter,startInd <,Count>*)
Example
[j1,j2]=[i1,i2,i3].**%fieldDel**('-',5,2)
7. Determine the number of fields in a string.
RefData.**%fieldCount**(delimiter)
Example
[j1,j2]=[i1,i2,i3].**%fieldCount**('-')

Actions on lists.

1. Get *Count* Elements of the list starting with *startInd*
refData.**%listGet**(*startInd <,Count>)*
example [j1,j2]=[i1,i2,i3].**%listGet**(5,3)

2. add to the list another list
refData.**%listPush**()
refData.**%listPush**
Example
[i1,i2,i3].**%listPush**()=[j1,j2]
[i1,i2,i3].**%listPush**=[j1,j2]

3. Take from the *Count* list node elements of the list
refData.**%listPop**(*Count)*
Example [j1,j2]=[i1,i2,i3].**%listPop**(2)

4. insert into the list instead *startInd* element another list
refData.**%listIns**(*startInd*)
Example
[i1,i2,i3].**%listIns**(7)=[j1,j2]

5. replace in the list *Count* elements starting from *startInd* element
refData.**%listSet**(*startInd <,Count>*)
Example
[i1,i2,i3].**%listSet**(4,3)=[j1,j2]

6. remove from the list *Count* elements starting from *startInd* element
refData.**%listDel**(*startInd <,Count>*)
Example
[j1,j2]=[i1,i2,i3].**%listDel**(5,2)
7. Determine the number of items in the list.
RefData.**%listCount**
Example
[j1,j2]=[i1,i2,i3].**%listCount**

Actions on the data array.

1. Add an additional element to the array.
 RefArr.**%arrPush**
eg
 Set m$.**%arrPush**=ABC
 Set m$.**%arrPush(**ABC,RT,125)
 Set $1=m$.**%arrPush(**ABC,RT)
 RefArr - is a shortcut to an array.
 ABC,RT,125 – added items.
2. Remove the element from the top of the array
 RefArr.**%arrPop**
eg
 [J1, j2] = m$.**%arrPop**
Or remove from the top element
 m$.**%arrPop**

13. Vectors.

The data can be stored integer vectors. The number of components of the vector depends on the dimension of the component.
The dimension of the vector components can be 64 bits, 32 bits, 16 bits and 8 bits. Accordingly, the dimension of these vectors will be 2, 5, 11 and 22. The vectors can be either signed or unsigned. Counting coordinates of the vector is from 0.

64 bit vectors.

Appeal to the sign component of the vector:
 refData.**%v64**(n);
n is the number of coordinates of the vector.
Appeal to unsigned vector component:
 refData.**%vu64**(n);
n is the number of coordinates of the vector.
The dimension of the vector is 2.

32 bit vectors.

Appeal to the sign component of the vector:
 refData.**%v32**(n);
n is the number of coordinates of the vector.
Appeal to unsigned vector component:
 refData.**%vu32**(n);
n is the number of coordinates of the vector.
The dimension of the vector is 5.

16 bit vectors.

Appeal to the sign component of the vector:
 refData.**%v16**(n);
n is the number of coordinates of the vector.
Appeal to unsigned vector component:
 refData.**%vu16**(n);
n is the number of coordinates of the vector.
The dimension of the vector is 11.

8 bit vectors.

Appeal to the sign component of the vector:
refData.**%v8**(n);
n is the number of coordinates of the vector.
Appeal to unsigned vector component:
refData.**%vu8**(n);
n is the number of coordinates of the vector.
The dimension of the vector is 22.

14. Functions library Win.

The **Win** Library allows you to create and manipulate the visual components of windows. This library is based on GTK + 3.0 and is closely connected with it. The properties of the **Win** library objects basically correspond to the GTK library's Widget properties.

The **Win** library must be initialized before use by the function:
%winInit()
After using the library resources should be released function:
%winFree()
The remaining functions are called between them.
Function:
%winNew(*type,nameObj<,nameParent,Arg...>***)**
Creates an object and places it in a hierarchy of visual objects.
type - the type of the object being created,
nameObj - the name of the new object,
nameParent - the name of the ancestor, may be empty if there is no ancestor,
Arg - parameters necessary to create an object. The parameter list depends on the object being created.
Function:
%winSetAtr(*typeAtr,nameObj, dataSet <, Arg ...>***)**
Sets the property of the object to the dataSet value.
typeAtr - the type of attribute to be changed,
nameObj - the name of the object,
dataSet is a new attribute value
Arg - additional parameters required when changing an attribute. The parameter list depends on the object being created.

Function:
%winGetAtr(*typeAtr,nameObj <, Arg ...>***)**
Read object attribute.
typeAtr - the type of attribute to get,
nameObj - the name of the object,
Arg - additional parameters required to access the attribute. The parameter list depends on the object being created.

Function:
%winIniToAtr(*typeAtr,nameIniFile<,nameIniFile...>***)**
Reads the Ini file and for the objects listed in this file for the attribute, sets the corresponding value specified in this file.
nameIniFile - the name of the Ini file
typeAtr - attribute whose value is to be set.

Function:
%winAtrToIni(*typeAtr,nameIniFile***)**
Writes to the Ini file for the objects listed in this file and the specified attribute value of the corresponding attribute of the object.
nameIniFile - the name of the Ini file
typeAtr - attribute whose value is to be set.

If you need to communicate with the server, then you need to open the communication channel.
Function:
%winOpenChannel(*iniPipe***)**
iniPipe is a file containing channel settings.
To display win components, you need to start the main application loop with the function:
%winStart(*nameObj***)**
nameObj is the main application window.

15. Module structure.

Module consists of the strings. the string can start from the tag. Then after the tag there is a colon(two spot). There should not be space between tag and a colon(two spot). The commands are separated from each other by semicolon';'

16. Objects.

In no objects MUMPS. But the PLO received widespread. Most of the modern programming languages anyway support the PLO. MSH is no exception. To make it OOP language must have to start the objects. In general, the description of the object consists of a recital and a part of the implementation. MSH does not support the declaration of variables, then the declarative part of the description of objects will not. There is a description of the object as part of MSH implementation. In MUMPS program code presented in the form of software modules. Entry points in the module are subroutines and functions. MSH in the software module and taken as a description of the object (class). Tags module is the module entry points. Entry points in this case are public get the properties of this class. Set public properties of this class will be the same as the entry point and get modified by symbol. The character "." Point is selected as such in the MSH. At properties in MSH access except get and set there is an operation Deleting Kill properties. In this case, the name of the properties modified by the prefix "..". Inheriting classes MSH is plural and implemented using **Parent** command. The classes listed in this command in the order are the ancestors of the class. During the execution of the program when accessing the property of an object description is first searched in the source module and then the modules listed in the **Parent** command in the order they appear. Earlier module position in the team gives it a higher priority. The protected properties are realized using the system property %this. Objects in MSH may be only a tree. Only primitive data types can be stored in the array.

Class methods are also the entry points in a class module. Creates an object using the standard features **%objNew**.

For example, we have a Person class with the property Age. The class module has a «Age» entry point and «.Age».

The variable [1,2] create an object.

. Set [1,2].**%objNew** = Person;

Title to the property of the Person object Age at entry.

Set [1,2] .Age = 50;

Title to the property Age Person object in reading.

The Set $1 = [1,2] .Age;

These commands compiler converts to access the Subprogramme. But this is not the only way to access the properties. and can be accessed via the method of recording to the properties Age.

Do [1,2] .Age (50);

The MSH unit performs a dual role. In addition to the class module, it can be interpreted as a program module. In this case, the entry points are treated as a sub-program. Suppose Person has a module ABC tag. Then it is possible to address how to program. The truth in this case, the variable %this will be empty.

Do Person.ABC (125, D, 25.6);

So you can organize class methods.

Modules can serve as object Type(class). Module name corresponds to object type and should not exceed 18 bytes. Property name by reading should be a tag in the module. Property by record should be the tag that is coincident with property name with prefix '.' .Heritance is provided by the **Parent** command. Any variable can become an object, to this effect it is enough to assign it to the predefined property **%objNew** is the name of the module type. For example there is a (module) type Org and property of this type Arg.
Let's create object of this type:
Set [An,12].**%objNew**=Org; then addressing to the Arg property by recording and by reading:
[An,12].Arg=[An,12].Arg+1;

17. Remarks.

Commands that have no arguments may not have an end-of-command indication «;». These commands are: **If, Else, End, Case, Break.**

18. Summary.

I believe that described language will serve desired purpose. Above listed great number of commands allow to implement modern concepts of programming. Availability of the task creating commands and event processing allows to simplify the process of programming. Data management in this language allows to exclude great amount of the errors by programs developing.

Author: Michael A. Sharymov. Email: misha_shar53@mail.ru