MSH programming language concepts.

Content.

Introduction.

The MSH language is built on the concepts of the MUMPS language. MUMPS is a little-known language developed in the last century. But it is still used in information applications. Information about this language is present on the Internet. There are working implementations of this language and a community of programmers supporting it. MUMPS are being developed in the USA and Russia. In addition, it is used, as far as I know, in Latin America, Germany, Australia, and China. In general, this living concept of language. When meeting with MUMPS, its archaic nature is striking. This development is designed to eliminate its shortcomings while preserving its advantages simplicity, consistency, data organization.

Organization of the program.

The translation unit is the MSH language module. The module starts from the standard 8 bytes identifying the language version. At the beginning of the line may be a label or spaces. The label ends with a “:” and is separated from the rest of the commands by any number of spaces. Lines consist of commands. A sign of the end of the team is the symbol “;”. The command is separated from the arguments by spaces. The case of characters does not matter. The command can be written with characters of any register. In addition, many teams have an abbreviated form. A command may have a condition for its execution. If there is one, then the symbol “?” Follows the command without spaces and the condition for the execution of this command. If the condition for executing the command is not equal to 0, then the command is executed. Inside the condition, spaces are unacceptable; otherwise, they will be treated as separators between the command and arguments.

For example:

SET? X> 5 Val [1] = 25; //correctly

SET? X> 5 Val [1] = 25; // syntax error

SET? (X> 5) Val [1] = 25; // right, inside () spaces are allowed

Arguments are separated by a ",". Inside arguments, a space is not a special character and can be contained anywhere. Typically, a command can have any number of arguments. Labels in the module are localized inside the module and must be unique in it, with the exception of labels inside the **CASE** command. Labels of the **CASE** command localized inside this command should be unique only inside this command and can duplicate labels both outside this command and in nested **CASE** commands.

Runtime.

At run time, an application has one or more tasks. All tasks are performed in parallel. In each task, programs are executed sequentially. At each moment in time, only one program code is executed in the task. The task ends with the end of the last running program. The main task is launched by the language runtime. The remaining jobs are generated by the **Job** command.

Data management.

For those who are familiar with the MUMPS language, the organization of data in MSH is quite clear. There is no description of the data in MSH. There is no data declaration. Data can be stored in the form of a tree, then access to the tree nodes is performed by an optional name and index. The index is enclosed in square brackets []. The name is in front of them.

For example:

SET Pr [4,5, “rt”] = Is [“ty ^ 578”];

Here **Pr** and **Is** are the names of trees. **4,5, “rt”** and **“ty ^ 578”** are node indices.

The tree can have an arbitrary number of levels and, accordingly, the index has the corresponding number of fields separated by commas. The index field can have an arbitrary value of the base type. The basic types in MSH are numbers and strings. Only nodes to which the recording was made are directly stored in the tree. The name, index field, and index itself can be an expression. After calculation, the name can only be an identifier.

Also, the data can be stored as a continuous array, then an access to the array element consists of an optional symbol name "$" and an index. An index is an integer. The smallest array index is 1. The name and index can be expressions. After calculation, the name can only be an identifier, and the index an integer. The array can be filled in any order, but if you wrote in **mc$1000**, then an **mc** array of **1000** elements will be created. Undefined elements will not contain values, but they will be present. The number of elements in the array can be found by referring to the zero element of this array.

For example: **mc$0**

The size of the array can be changed by writing a new array length to this element. But in the general case, this is not necessary, since the array expands automatically.

Tree nodes and array elements contain data of basic types. These are either strings or numbers. How programmer data is stored is not concerned. The storage of data types and their manipulation is the responsibility of the MSH language implementation.

Data localization.

MSH data is divided into global and local. They differ in the type of name. Global data is stored in long-term memory and does not depend on the lifetime of the application. Once they are created, they can be changed by the application and will exist until the application destroys them with the KILL command. All global data has the prefix “^” in the name.

Access to global data is simultaneously possible from various tasks. Therefore, when accessing globals, synchronization is necessary. Such synchronization is always automatic. There is always a synchronization primitive in the global descriptor and it controls access to the global. Moreover, when reading the global is blocked by reading, when writing to the global, it is blocked by writing. No additional globals synchronization is required. Accesses to global arrays are also synchronized.

For example:

^gl[87,9] - access to the global tree node.

^glar$45 - access to the element of the global array.

Local data exists only while the application is running. The next application launch cannot access the local data of the previous launch.

The scope of local data depends on its type. There are three types of data localization.

1. Local program data. They are localized inside the program and exist from the moment the program is launched until its completion. If a program calls a subprogram, then new subprogram data is created, and the local program data inside the subprogram is not visible. When you return to the program, the local program data becomes available again. Local program data does not have a name.

For example:

[7,9] - access to the node of the tree localized inside the program.

$5 - access to an array element localized inside the program.

There is one exception. An array of parameters passed to the **A$** program is also localized inside the program.

2. Local application data. They are visible in all tasks of the application. You can contact them from any task. They exist from the moment they are created in the application by any task until the application is completed or until they are destroyed by the **KILL** team. Names of such data are prefixed with "%". These variables are simultaneously available in different tasks, so they are as synchronized as globals.

For example:

%dapp[87.9] - access to the tree node localized inside the application.

%dapp$45 - access to an array element localized inside the application.

3. Local job data. They are localized inside the task and exist from the moment they are created in any task program until the task is completed or destroyed by the KILL team. Such data must have a name and do not contain the prefixes “^” and “%”.

The exception is the array of program parameters **A$**, it is localized inside the program.

For example:

djob[87,9] - access to the tree node localized inside the job.

djob$45 - access to an array element localized inside the job.

Access to variables can only have the types listed here.

Constants.

Basic data types can be either numeric or string. The numeric form is either an integer or a real number in the presence of a decimal point. The base of the numbers is 10. They can be either positive or negative.

For example:

25, -4, 789.56, -9.3

String constants are any sequence of characters. If a constant consists only of letters and numbers, then you can not enclose it in quotation marks, since it cannot be confused with a variable. If the constant contains other characters, then it must be enclosed in either single or double quotes.

For example:

"Rt@tty#123"

`14“ 5 * 7 ”89 \?`

125Dsv

Features of some teams.

CONSTANT command.

You can give a constant a name using the CONSTANT command. These are broadcast time names. Broadcast replaces them with their values. At run time, these names no longer exist. Therefore, you cannot assign an expression to a name in the CONSTANT command. The value must be exactly a constant. The names of the constants must be chosen so that they do not coincide with the values of the constants specified in the program without quotes.

For example:

Constant ioInOut = "/ini/par.ini",maxIs=25;

The ioInOut and maxIs constants are assigned values. Further in the program these names can be used instead of these values.

The **Constant** team has also 2 form. In this case, the right side of the equality is absent. The meaning of this team is different. The name is the name of the module containing the constants. The constants of this module are exported to the current module. The module importing constants does not contain any additional descriptions about import. The importing module can contain both constants and programs only.

For example:

Constant sysCnsNet, usrCnsByx;

sysCnsNet and **usrCnsByx** are module names containing constants.

Both forms can occur as arguments to the same **Constant** command.

XECUTE command.

The **XECUTE** command is a rather exotic team, but it is present in the MUMPS language. In other programming languages, she met me only in JavaScript. There it is called eval. When this command is executed, the expression of the argument of this command is calculated, and then this expression is converted to a string, interpreted as an MSH command, and executed.

For example:

XECUTE "SET $1= 89;";

As a result, the variable $1 will receive the value 89.

This is a primitive example, in this form it hardly makes sense to use this command. The arguments to the **XECUTE** command are expressions, which allows you to generate various MSH commands at the time the program runs.

The command is executed in the context of the program where the command is located. She has access to all program resources, including local program data.

Commands COPY and MOVE.

The **COPY** command is similar to the **MERGE** MUMPS command. These commands copy the source node along with all descendants to the receiver node. The argument of these commands consists of two fields separated by the symbol "=". To the right of this sign is the source node, to the left is the receiver. An abbreviated link is allowed as nodes, in this case the whole tree is used. These commands copy not only the descendants, but also the nodes themselves. The **COPY** command performs data merging. The source data is copied to the receiver without cleaning it. The source does not change. The **MOVE** command actually moves the data. First, the receiver is cleaned, then all the descendants of the source node are copied and the source node is deleted with all its descendants.

For example:

// node us [5,6] is copied to node [1]

COPY [1]=us[5,6];

// the whole us tree moves to the node [8]

MOVE [8]=us[];

These commands can also be used to copy arrays. In this case, only shortened links can be used as the source and receiver.

The program arguments are copied to the arg array.

COPY arg$=A$;

You can use the **MOVE** command to move.

MOVE a1$ = b1$;

You can copy and move any arrays.

Resource synchronization.

When performing several tasks, you need access to shared resources. Synchronization is performed by lock commands. Synchronization commands by themselves do not block anything, this is a set of agreements that allow to differentiate access to shared resources. If the lock commands are not shared across jobs, no access synchronization will occur. Synchronization is built on lock names. These names are localized within the application and are the same in all tasks. In locks, the concept of many readers and only one writer at a time is accepted. Accordingly, there are read locks and write locks.

The **LockR** command blocks read names. The names listed in her arguments will be blocked by reading. Any number of tasks can block the same names, but a task that tries to block any of these names will wait for all these names to be unlocked. After capturing the write name, no read lock command can be executed until the write lock releases the name. If it is not possible to lock, the command will wait for the name to be released.

For example:

LockR name1, name2, name3;

These names will be blocked by reading. Another task can also block these names without waiting for them to be unlocked by reading.

The **LockW** command locks names by record. The names listed in her arguments will be blocked by record. If the names listed in the arguments are already blocked by any lock command, then this command will wait for the release of these names.

For example:

LockW name1, name2, name3;

These names will be locked by record.

The **LockUn** command unlocks this name. If the name is blocked for reading several times, then you need to unlock the same number of times.

For example:

LockUn name1, name2, name3;

The standard functions have analogues of these commands with a timeout.

The abbreviated form of the SET command.

The **SET** command has an abbreviated form. In this case, the left side of the equality is absent; its role is played by the last variable mentioned in the expression.

For example:

SET $ 1 = 2, $ 1 + 3;

The variable $ 1 will become equal to 5.

If there are several variables in the expression, the result will be assigned to the last variable.

For example:

SET $ 1 = 1, $ 2 = 2, $ 3 = 3, $ 1 + $ 2 + $ 3;

The $ 3 variable will become 1 + 2 + 3 = 6. Although this form is more appropriate to use only in very simple cases, similar to the first example. The second example is provided only as an illustration of the capabilities of this form of the **SET** command.

Block commands.

Block commands form a block of commands and simultaneously serve as the heading of the block. Each block command must have its own **END** command, even if there is only one command in the block.

IF command.

The **IF** command forms a block that is executed if the conditions for the execution of this command are true. This command has no arguments. This block may contain **ELSE** commands. The **ELSE** command has no arguments. Outside the **IF** block, these commands have no meaning and cannot be applied. If there are **ELSE** commands in the **IF** block when the condition of the **IF** command is met, only the commands behind the **IF** command are executed until the next **ELSE** command. An **ELSE** command may contain an execution condition, then if true, only commands located after this command until the next **ELSE** or **END** command will be executed in this condition. An **IF** block can contain only one **ELSE** command without a condition for its execution, and it must be the last among the **ELSE** commands.

For example:

IF? X <0;

SET y = 1;

ELSE? X <5;

SET y = 2;

ELSE? X <10;

SET y = 3;

ELSE SET y = 4;

END;

The condition for the execution of this command may be absent, then this block will be executed in any case. Although it is difficult to imagine why this could be useful.

CASE command.

The semantics of this command are somewhat different from the semantics of the other MSH teams. The condition for executing a command in it is not such. In the condition of execution of the **CASE** command, the expression should calculate the label to which control will be transferred. This command has no arguments.

Each label in this command forms a block from this label to the next. When transferring control to a label, only commands are executed until the next label, and then the current **CASE** block is exited. This command is closer to Pascal notation than to the C command. If as a result of calculating the name of the label a name is found that is absent in the current **CASE** block, the commands located after the **CASE** command before the first label will be executed.

```
CASE?L_$J; //compute label
  SET x[1]=1; // if the label is not found then the commands of this block are executed
  SET a[2]=x[1]+1;
L1:  SET x[1]=2; // label command block L1
  SET a[2]=x[1]+2;
L2:  SET x[1]=3; // L2 label command block
  SET a[2]=x[1]+3;
END;
```

Labels in this command implicitly form an internal block of commands. After the execution of such a block of commands, control is transferred outside the block of the **CASE** command.

WHILE command.

The **WHILE** command is used to organize the loop. The condition for the execution of this command sets the condition for the continuation of the loop. As long as the condition for executing the command is not 0, the block formed by this command will be executed. The block ends with the **END** command. The **END** command of such a block has a feature. It may have a block termination condition. If the condition is not 0, then the block will be completed. Moreover, these conditions may be present both in the **WHILE** command and in the **END** command.

For example:

```
WHILE? X[7]> 0; // condition to continue the loop
 SET y[2] = x[7] +2;
 BREAK?Y[2]<0;
 SET x [7] = x [7] +1;
 END? X [7]> 20; // condition for ending the loop
```

Inside the block, the loop can be interrupted by the **BREAK** command.

Block loop iterators.

Block loop iterators are optimized for access to tree nodes. They use an internal link to optimize access to crawl nodes. This imposes a restriction on the use of block iterators. Inside the block, you cannot change the structure of the tree. You cannot write to this tree. In block loop iterator commands, 2 arguments.

The first argument is required, the link to the node whose descendants will be bypassed. Reference index. Shortened links may be used. The second argument is the link to the node where the descendant index will be stored. This argument is optional. Inside the block, the 2nd argument should not change. Expressions like $$2 or [[3]] are not allowed if $2 or [3] is changed inside the iterator block. Changes to these variables will not be taken into account. Access to the descendant index can be obtained through the **%queryKey** system variable, and the node data from the **%queryData** property. If the descendants must not be bypassed from the beginning, then the 2nd argument is required and the node index must be placed in it, followed by the descendants. If there is a second argument, but you need to go around from the beginning, then before the loop you need to delete this variable with the **KILL** command.

Commands may have a block execution condition. This condition is checked only once upon entering the block.

You can go around not only the nodes of the tree, but arrays. In this case, the serial number of the array field gets into the second argument. Only the shortened link can be used as the first argument.

NEXT command.

The **Next** command iterates over the immediate descendants of the tree node. Descendants do in the forward direction from the minimum to the maximum index.

For example:

```
NEXT us[4,5]; // 2 argument is not specified, the index is taken
    // from system variable% queryKey
    SET $1 =%queryKey, $2=%queryData;
    END?$1>1000; // condition for ending the loop
```

The node index is immediately placed in the data.

```
KILL $1;
NEXT us[4,5],$1;
    SET $2=%queryData;
    END;
```

An abbreviated link is used as a reference node. In this case, the first level of the tree is bypassed.

```
KILL $1;
NEXT us[],$1;
    SET $2=%queryData;
```

END;

Crawl after index 3.

SET $1= 3;

NEXT us[4,5],$1;

SET $2=%queryData;

END;

When traversing an array, all fields are circumvented in order, even those in which data is not defined.

For example:

KILL $1;

NEXT us$,$1;

SET $2 =%queryData;

END;

Inside the block, the loop can be interrupted by the **BREAK** command.

BACK command.

The **BACK** command differs from the **NEXT** command only in the traversal direction from the last vertex to the first.

For example:

KILL $1;

BACK us[4,5],$1;

SET $2 =%queryData;

END;

QUERY command.

The **QUERY** command traverses all descendants of a node from left to right and from top to bottom to full depth in the forward direction. The 2nd argument contains the entire additional index. If this index has more than 1 field, then the list is placed in the 2nd argument.

Otherwise, this command is similar to the **NEXT** command.

The **QUERY** command traverses the array only in the forward direction from left to right.

For example:

KILL $1;

QUERY us[4,5],$1;

SET $2 =%queryData;

END;

But this command bypasses only the vertices that matter.

Non-block commands Data tree traversal commands.

In these commands, both arguments to the command are required. The second argument stores the index, after which the next command will find the next vertex. In these commands internal links are not saved and therefore there is no restriction on adjusting the tree that is bypassed. For the same reason, access times to peaks can be much longer. Arrays can also be bypassed with these commands.

NEXT1 command.

The **NEXT1** command gives the next vertex at the same level below the reference node.

Node data is available in the **%queryData** system variable.

For example:

SET $1= 2;

NEXT1 us[1,4],$1; // will give the node located at level 3 after the node us [1,4,2]

BACK1 command.

The **BACK1** command gives the previous vertex at the same level below the reference node.

Otherwise, it is similar to the **NEXT1** command.

QUERY1 command.

The **QUERY1** command gives the next vertex of a tree branch while traversing the entire node from top to bottom and from left to right. The 2nd argument contains the entire additional index. If this index has more than 1 field, then the list is placed in the 2nd argument.

Otherwise, it is similar to the **NEXT1** command.

Program execution management.

A module label can be a call point for a program, function, a call point for a new task, an object property, an object method, and a label, depending on the access to this label.

Moreover, the label can be accessed differently in different parts of the module.

For example:

LB: Set Val[25] =7+A$1; Return Val[25];

Do LB(78); // Access to the program. The return value is ignored.

Set Val[7] = 8 * LB(6); // Calling as a function the return value is used.

JOB LB(17, “yt”); // call a new job

Set Val[9] = Mod.LB(3); // Access to the class method Mod- the name of the module in this context is interpreted as the name of the class of the object.

Set Val[15] = Obj[1,2, A].LB; // Access to the LB property of the object Obj[1,2, A],

Go LB; // Go to the label LB

In the above examples, in addition to 4 and 5, the call is carried out inside the current module. The call of programs and functions can be carried out to any available module. By the way, this also applies to the **Go** command.

For example, in the Mod module there is a label Lb. Then the appeal to it will look like:

Do Mod.Lb(78);

Set Val[7] = 8 * Mod.Lb(6);

Set Val[9] = Mod.Lb(3);

JOB Mod.Lb(78);

Go Mod.Lb;

In general, a label is used as a program or function. If the module is used as an object class, the label is a property or method of this object. The properties of an object and its methods are another form of access to programs and functions; therefore, everything said about programs and functions equally applies to properties and methods. There are no significant nuances when accessing objects only in terms of inheritance. This issue will be described in more detail below.

In the general case, the name of the module and the name of the label in the call to it are expressions and can be calculated at the time of call.

For example:

Set $1 = “Md“,$2 = “Lb“;

Do $1.$2;

The program terminates with the Return command; A module can be completed with the End command;

Passing parameters.

In MSH, parameters are passed only by value. In general, no references and pointers in terms of the C ++ language and the enormous problems associated with their use here, in principle, do not exist. A program always takes an arbitrary number of parameters in an A $ array. Their number can be found by referring to the 0 element of the array A $ 0. The programmer is responsible for the meaning of these parameters.

For example:

We have a Lb program. It can be addressed:

DO Lb(1,2,7);

DO Lb(25.7);

DO Lb();

JOB Lb(8);

SET [7.8] = Lb(187, “Privet”);

Not passed parameters in the program are not defined.

Passing parameters by value does not prevent you from passing variable names into the program and using them there. For example:

Set us[5] = 48;

Do Lb("us");

...

Return;

Lb: **Set par[8,1,9] = A$1[5];** //A$1 - the first parameter passed contains the name

// variable: us.

// [5] - access to the top of this tree

// as a result, the node us [5] = 48 will be placed in the par [8,1,9] node

Return;

As a result, the name of the variables can be manipulated arbitrarily.

When going to the label with the GO command, the parameter array does not change. MSH has a **GO** command form with new parameters passed. In this case, the current parameter array is replaced with the new one specified in the **GO** command call.

For example:

GO Lb(1,2,7);

Event handling.

Event processing is a powerful programming mechanism that is generally not included in high-level languages. But its presence in libraries and the operating system speaks of its importance. Libraries of visual components are built on this mechanism. The presence of event processing in the language expands the capabilities of the language. Opens a new event-driven programming style. By the way, such a respected programming language as Assembler has such features. Therefore, this mechanism has been added to the MSH language.

The basis of this mechanism is the event. It is localized inside the application. Accordingly, these names must be unique throughout the application. If the event name matches the name declared in the **CONSTANT** command, the event name will be replaced with the value of the declared constant. Be careful. When naming events, it is advisable to adhere to some kind of naming strategy. For example, assign event names starting with evu. The length of the name, taking into account the UTF8 encoding, should not exceed 18 bytes. This limitation is associated only with the current language implementation.

An event created in one task is visible and can be processed in any tasks. Event handlers can be any number and they can be in different tasks. After the event occurs, it is checked whether there are handlers for this event; if there are no handlers, the event is deleted. If there are handlers then they are executed sequentially. Events can be system and user. System events are generated by the system. A custom event is generated by the **EVENTTRAP** command. The event passes parameters to the handlers, as when invoking a program. After processing, event handlers are not deleted. To delete an event, use the **EVENTDELETE** command. An event can be handled by the **EVENTCALL** and **EVENTWAIT** commands.

These commands allow you to organize asynchronous execution of programs.

EVENTTRAP command.

The command creates a custom event. The command format resembles a program call, only the event name is used instead of the program name. Event names are local within the application, so they are visible in all tasks.

The command will create an event with the specified name. Event handlers will be passed the parameters of this command argument.

For example:

EVENTTRAP?X[1]>8 evuBoxData (us[7],$4),evuKorXY(us[X], us[Y],sh);

2 events evuBoxData and evuKorXY are generated as variables us[7],$4,us[X],us[Y] and the string constant sh.

If there are currently no handlers for this event, then the event is not generated.

EVENTDELETE command.

The command removes the event handlers listed in the program arguments.

For example:

EVENTDELETE?X[1]>7 evuKorXY,evuBoxData;

Events will be deleted in the order in the team.

EVENTCALL command.

The command assigns an event handler. A handler is a program. This program will be called asynchronously from the running task and parameters will be passed to it. After execution of the handler program, control will be returned to the main task at the interruption site.

For example:

EVENTCALL evuBoxData=Mod1.intEvuBoxData,evuKorXY=Mod2.intevuKorXY;

The program handler will be called in the new program context. The context of the task will correspond to the context of the task in which the **EVENTTRAP** command was executed.

EVENTWAIT command.

The team is waiting for events to occur. This command will block the current task until the events listed in its arguments occur. All events listed in its arguments should occur to continue execution of the current thread. The current program parameters are replaced by those transferred in the event creation command.

For example:

EVENTWAIT evuBoxData, evuKorXY;

Vectors.

Vectors are closely related to the current implementation. They store signed and unsigned integers. The dimension of these vectors depends on the bit depth of the vector component and it is fixed.

64 bit vectors.

The dimension of such a vector is 2. Components can be 64-bit integers or unsigned integers. Such numbers are stored in any variables.

The appeal to the iconic integer components of the vector.

SET us[5].%v64(0) = ss$1.%v64(1);

%v64 - access to the integer signed component of the vector.

An appeal to unsigned integer components of a vector.

SET us[5].%vu64(0)=ss$1.%vu64(1);

%vu64 - access to the whole unsigned component of the vector.

32 bit vectors.

The dimension of such a vector is 5. The components can be 32-bit integers or unsigned integers. Such numbers are stored in any variables.

The appeal to the iconic integer components of the vector.

SET us[5].%v32(0)=ss$1.%v32(4);

%v32 - access to the integer signed component of the vector.

An appeal to unsigned integer components of a vector.

SET us[5].%vu32(0) = ss$1.%vu32(4);

%vu32 - access to the whole unsigned component of the vector.

16 bit vectors.

The dimension of such a vector is 11. The components can be integer or unsigned 16-bit numbers. Such numbers are stored in any variables.

The appeal to the iconic integer components of the vector.

SET us[5].%v16(0) =ss$1.%v16(10);

%v16 - access to the integer signed component of the vector.

An appeal to unsigned integer components of a vector.

SET us [5].%vu16(0) = ss$1.%vu16(4);

%vu16 - access to the whole unsigned component of the vector.

8 bit vectors.

The dimension of such a vector is 22. The components can be integer or unsigned 8-bit numbers. Such numbers are stored in any variables.

The appeal to the iconic integer components of the vector.

SET us[5].%v8(0)=ss$1.%v8(21);

%v8 - access to the integer signed component of the vector.

An appeal to unsigned integer components of a vector.

SET us[5].%vu8(0) = ss$1.%vu8(21);

%vu8 - access to the whole unsigned component of the vector.

Operations.

Operations in MSH play a special role. It is they who control the conversion of data types. Depending on the operation, the operands are cast to the desired data type. The type of data result uniquely corresponds to the type of operation. String and numeric operations do not overlap, like the “+” operation in C-like languages. The type of operation in MSH does not depend on the type of operands, everything is exactly the opposite. There are no operations priorities in MSH; this is the historical legacy of MUMPS.

For example:

SET $1=2 + 3 * 4;

$1 will be 20, not 14.

In order for the result to be 14, parentheses are used.

SET $1=2 + (3 * 4);

As the operation of joining strings, the symbol "_" is used.

The lack of priority of operations is unusual, but quite convenient. The reference to the naturalness of priorities, when operations are more than add up to multiply becomes very doubtful. Having once learned that there are no priorities, there is no need to painfully recall their priorities, and if something goes into the documentation. In general, this is a matter of habit.

Objects.

The presence of objects in modern programming languages is a good form. In the general case, objects consist of 2 parts. Parts of the declarative description and parts of the implementation. In MUMPS systems, variables do not have the declarative part of a type declaration. Classes are basically user data types. In order not to violate the principles of MUMPS, MSH lacks the declarative part of the class description. And as it turned out, you can perfectly do without it. Only part of the implementation is left of the class. The implementation of the class can be perfectly represented by the standard module. Only for the class had to introduce additional conventions associated with the description of the properties of the object. Objects can only be in the tree. Placing an object in an array will fail because there is no place to store the properties of the object. Although if the object does not have properties, then you can try. Although what kind of object it is.

To describe a public property, you need a function that returns the value of this property. It must have a property name and be in the module with the class name. A program to write to a property. This program has a property name prefixed with ".". And a property removal program. This program has a property name prefixed with "..".

The function name in the module may correspond to the public property for reading. In this case, the return value of such a function is passed to the calling program as the value of the public property.

The constructor of the class is the **%objNew** data method. If, when creating an object, it is necessary to determine any properties or obtain resources, then any class module program (class method) can be used. But it is advisable to adhere to a strategy for naming class constructors. For example, the constructor name must match the class name.

Access to the protected properties of the class is carried out through the system property% this.

The destructor is the removal of the object using the **KILLD** command. If you need to free any resources or perform additional manipulations, then this can be done by any program of this class (class method). As in the case of the constructor, it is advisable to adhere to some naming strategy when naming the destructor.

For example:

```
// Class Person

// Property Age

// read the public property Age
Age:    RETURN [%this,Age];

        // record public property Age
.Age:   SET [%this,Age] =A$1;

        Return;

        END;
```

An appeal from programs to an object and its public properties will look as follows.

```
        // create a Person object
```

SET us[1,2].%ObjNew=Person;

// write the value 50 to the Age property

SET us[1,2].Age=50;

// read the Age property

SET u$1 = us[1,2].Age + 5;

// delete the Age property

KILL us[1,2].Age;

Inheritance of objects.

MSH classes support multiple inheritance. The **PARENT** command sets all the ancestors of this class. Moreover, the order of the names of the ancestors of this class in the command arguments determines the priority of inheritance. The later the class is mentioned, the lower its priority.

For example:

PARENT USER,BOX;

The class inherits from the ancestors of **USER** and **BOX**. **USER** ancestor priority is higher. What is the priority. When accessing an object, the public property or method of the class will be searched in the class itself, if they are not found there, then they will be searched in the ancestor with the highest priority, and then in the ancestors of this class and so on by priority.

Sharing with a file.

In MSH, file sharing is organized at the most primitive level. Files in MSH play a supporting role. The exchange is organized only with text files. The file structure is text fields separated by a specified separator. The delimiter is in the **%dlmIO** system variable. By default, this variable is ",". It is read-writing. When writing to a file, the variables are converted to a string type and written to the file through the delimiter. When reading from a file, variables are selected through a separator and are brought to a normalized form. If the field is a record of a number, then the number is placed in the variable. Exchange with the file occurs through the array B$. When writing, the array B$ is written to the file through the delimiter. When reading from a file, the fields are selected into the B$ array.

File exchange commands take the file path as arguments.

For example:

From an array of B$ data is written to a file. The file is opened by writing. The data in the file is replaced.

Write "txt / tt1.txt";

Data is read into the B$ array. The array is pre-cleared.

Read "txt / tt1.txt";

txt / tt1.txt - path to the file.

Conclusion.

This document is not a substitute for the MSH language description, but only supplements it. Here, not all the features of the MSH language are considered, but only those to which I would like to draw your attention.

Author: Sharymov Mikhail Alekseevich. Email: misha_shar53@mail.ru