Концепции языка программирования MSH.

Оглавление

Введение.

Язык MSH построен на концепциях языка MUMPS. MUMPS мало известный язык разработанный в прошлом столетии. Но он до сих пор используется в информационных приложениях. Информация об этом языке присутствует в интернете. Есть рабочие реализации этого языка и сообщество программистов поддерживающих его. Разработкой MUMPS занимаются в США и России. Кроме того используется он, насколько мне известно, еще в странах Латинской Америки, Германии, Австралии, Китае. В общем эта живая концепция языка. При знакомстве с MUMPS бросается в глаза его архаичность. Данная разработка призвана устранить его недостатки сохранив его достоинства простоту, логичность, организацию данных.

Организация программы.

Единицей трансляции является модуль языка MSH. Модуль начинается из стандартных 8 байт идентифицирующем версию языка. В начале строки может находится метка либо пробелы. Метка заканчивается символом «:» и от остальных команд отделена любым количеством пробелов. Строки состоят из команд. Признаком конца команды является символ «;». Команда отделена от аргументов пробелами. Регистр символов имени команды значения не имеет. Команда может быть записана символами любого регистра. Кроме того многие команды имеют сокращенную форму. Команда может иметь условие ее выполнения. Если таковое имеется, то за командой без пробелов следует символ «?» и условие выполнения данной команды. Если условие выполнения команды не равно 0, то команда выполняется. Внутри условия пробелы недопустимы иначе они будут оттрактованы как разделители между командой и аргументами.

Например:

SET?[5]>5 Val[1]=25; //правильно

SET?[2] > 5 Val[1]=25; //ошибка синтаксиса

SET?([1,2] > 5) Val[1]=25; //правильно, внутри () пробелы допустимы

Аргументы разделены символом «,». Внутри аргументов пробел не является спец символом и может содержаться в любом месте. Как правило команда может иметь любое количество аргументов. Метки в модуле локализованы внутри модуля и должны быть в нем уникальны, за исключением меток находящихся внутри команды **CASE**. Метки команды **CASE** локализованы внутри этой команды должны быть уникальны только внутри этой команды и могут дублировать метки как вне этой команды, так и во вложенных командах **CASE**.

Среда выполнения.

Во время выполнения приложение имеет одно или более заданий. Все задания выполняются параллельно. В каждом задании программы выполняются последовательно. В каждый

момент времени в задании выполняется код только одна программа. Задание завершается с окончанием последней выполняемой программы. Главное задание запускается средой выполнения языка. Остальные задания порождаются командой **Job**.

Управление данными.

Для знакомых с языком MUMPS организация данных в MSH достаточно понятна. Описание данных в MSH отсутствует. Никакого декларирования данных нет. Данные могут храниться в виде дерева, тогда обращение к узлам дерева выполняется по необязательному имени и индексу. Индекс заключается в квадратные скобки [ ] . Имя находится перед ними.

Например:

SET Pr[4,5,«rt»]=Is[«ty^578»];

Здесь **Pr** и **Is** -имена деревьев. **4,5,«rt**» и **«ty^578»** индексы узлов.

Дерево может иметь произвольное количество уровней и соответственно индекс имеет соответствующее количество полей разделенных запятыми. Поле индекса может иметь произвольное значение базового типа. Базовыми типами в MSH являются числа и строки. В дереве непосредственно хранятся только узлы в которые производилась запись. Имя, поле индекса и сам индекс могут быть выражением. После вычисления имя может быть только идентификатором.

Также данные могут храниться в виде непрерывного массива, тогда обращение к элементу массива состоит из необязательного имени символа «$» и индекса. Индекс это целое число. Наименьшим индексом массива является 1 Имя и индекс могут быть выражениями. После вычисления имя может быть только идентификатором, а индекс целым числом. Массив можно заполнять в любом порядке, но если вы записали в **mc$1000**, то будет создан массив **mc** из **1000** элементов. Не определенные элементы не будут содержать значений, но они будут присутствовать. Количество элементов в массиве можно узнать обратившись к нулевому элементу данного массива.

Например: **mc$0**

Размер массива можно изменить записав в этот элемент новую длину массива. Но в общем случае в этом нет необходимости, так как массив расширяется автоматически.

Узлы дерева и элементы массива содержат данные базовых типов. Это либо строки, либо числа. Как хранятся данные программиста не касается. За хранение типов данных и манипуляции ими отвечает реализация языка MSH.

Локализация данных.

Данные в языке MSH делятся на глобальные и локальные. Отличаются они типом имени. Глобальные данные хранятся в долговременной памяти и не зависят от времени жизни приложения. После того как они созданы, они могут меняться приложением и будут существовать до тех пор пока приложение их не уничтожит командой **KILL.** Все глобальные данные имеют в имени префикс «^».

Доступ к глобальным данным одновременно возможен из различных заданий. Поэтому при обращении к глобалям необходима синхронизация. Такая синхронизация выполняется автоматически всегда. В дискрипторе глобали всегда есть примитив синхронизации и он управляет доступом к глобали. Причем при чтении глобаль блокируется по чтению, при записи в глобаль она блокируется по записи. Никакая дополнительная синхронизация глобалей не требуется. Обращения к глобальным массивам так же синхронизировано.

Например:

^gl[87,9] — обращение к узлу глобального дерева.

^glar$45 — обращение к элементу глобального массива.

Локальные данные существуют только пока выполняется приложение. Следующий запуск приложения не может получить доступ к локальным данным предыдущего запуска.

Область существования локальных данных зависит от их вида. Есть три вида локализации данных.

1. Локальные данные программы. Они локализованы внутри программы и существуют от момента запуска программы до ее завершения. Если программа вызывает подпрограмму, то создаются новые данные уже подпрограммы, а локальные данные программы внутри подпрограммы не видны. При возвращении в программу локальные данные программы снова становятся доступными. Локальные данные программы не имеют имени.

Например:

[7,9] — обращение к узлу дерева локализованного внутри программы.

$5 — обращение к элементу массива локализованного внутри программы.

Есть одно исключение. Массив параметров переданных программе **A$** локализован тоже внутри программы.

2. Локальные данные приложения. Они видны во всех заданиях приложения. Из любого задания к ним можно обращаться. Они существуют от момента создания их в приложении любым заданием до момента завершения приложения либо до момента их уничтожения командой **KILL**. Имена таких данных имеют префикс «**%**». Эти переменные одновременно доступны в разных заданиях, поэтому они так же синхронизированы как и глобали.

Например:

%dapp[87,9] — обращение к узлу дерева локализованного внутри приложения.

%dapp$45 — обращение к элементу массива локализованного внутри приложения.

3. Локальные данные задания. Они локализованы внутри задания и существуют от момента их создания в любой программе задания до момента завершения задания или уничтожения командой **KILL**. Такие данные обязательно имеют имя и не содержат префиксов «^» и «%». Исключением является массив параметров программы **A$**, он локализован внутри программы

Например:

djob[87,9] — обращение к узлу дерева локализованного внутри задания.

djob$45 — обращение к элементу массива локализованного внутри задания.

Обращение к переменным может иметь только перечисленные здесь виды.

Сокращенный синтаксис.

Никакого отношения к аналогичному термину в языке MUMPS, этот термин не имеет. Сокращенный синтаксис в языке MSH используется для обращения либо ко всему дереву, либо ко всему массиву. Он применяется только в отдельных командах и там где он допустим это всегда оговаривается. Обращение ко всему дереву состоит из имени и обязательных квадратных скобок. Для локального дерева программы имя не указывается.

Например:

us[ ] - обращение ко всему дереву us.

[ ] - обращение к локальному дереву программы.

Обращение ко всему массиву состоит из имени и обязательного символа «$». Для локального массива программы имя не указывается.

Например:

us$ - обращение ко всему массиву us.

$ - обращение к локальному массиву программы.

Константы.

Базовые типы данных могут иметь числовой вид либо строковый. Числовой вид это либо целое число, либо действительное при наличии десятичной точки. Основанием чисел является 10. Они могут быть как положительными, так и отрицательными.

Например:

25, -4, 789.56, -9.3

Строковые константы представляют из себя любые последовательности символов. Если константа состоит только из букв и цифр, то ее можно не заключать в кавычки, так как ее нельзя спутать с переменной. Если константа содержит другие символы, то она должна быть заключена либо в одинарные, либо в двойные кавычки.

Например:

«rt@tty#123»

`14«5*7»89\?`

125Dsv

Особенности некоторых команд.

Команда CONSTANT.

Константе можно дать имя с помощью команды **CONSTANT**. Эти имена времени трансляции. Трансляция заменяет их на их значения. Во время выполнения этих имен уже не существует. Поэтому в команде **CONSTANT** имени нельзя присвоить выражение. Значение должна быть именно константа. Имена константам надо выбирать так, чтобы они не совпадали со значениями констант, указанными в программе без кавычек.

Например:

Constant ioInOut= «/ini/par.ini»,maxIs=25;

Константам ioInOut и maxIs присваиваются значения. Далее в программе эти имена можно употреблять вместо этих значений.

Команда **Constant** имеет и 2 форму. В этом случае правая сторона равенства отсутствует. Смысл этой команды иной. В качестве имени используется имя модуля содержащего константы. Константы этого модуля экспортируются в текущий модуль. Модуль импортирующий константы никаких дополнительных описаний об импорте не содержит. Импортирующий модуль может содержать как только константы, так и программы.

Например:

Constant sysCnsNet,usrCnsByx;

sysCnsNet и **usrCnsByx** имена модулей содержащих константы.

Обе формы могут встречаться в качестве аргументов одной команды **Constant.**

Команда XECUTE.

Команда **XECUTE** довольно экзотическая команда, но она присутствует в языке MUMPS. В других языках программирования она встретилась мне только в JavaScript. Там она называется eval. В момент выполнения этой команды происходит вычисление выражения аргумента этой команды, а затем это выражение преобразуется в строку, интерпретируется как команда MSH и выполняется.

Например:

XECUTE «SET $1=89;»;

В результате переменная $1 получит значение 89.

Это примитивный пример, в таком виде эту команду навряд ли имеет смысл использовать. Аргументы команды **XECUTE** являются выражениями, что позволяет генерировать различные команды MSH в момент выполнения программы.

Команда выполняется в контексте той программы где находится команда. Ей доступны все ресурсы программы, в том числе локальные данные программы.

Команды COPY и MOVE.

Команда COPY аналогична команде MERGE языка MUMPS. Эти команды копируют узел источник вместе со всеми потомками в узел приемник. Аргумент этих команд состоит из двух полей разделенных символом «=». Справа от этого знака находится узел источник, слева приемник. В качестве узлов допустима сокращенная ссылка, в этом случае используется все дерево. Эти команды копируют не только потомков, но и сами узлы. Команда COPY выполняет слияние данных. В приемник без его очистки копируются данные источника. Источник не меняется. Команда MOVE выполняет фактически перемещение данных. Предварительно приемник очищается, затем происходит копирование всех потомков узла источника и узел источник удаляется со всеми своими потомками.

Например:

```
//узел us[5,6] копируется в узел [1]
COPY [1]=us[5,6];
//все дерево us перемещается в узел [8]
MOVE [8]=us[];
```

Эти команды могут использоваться и для копирования массивов. В этом случае в качестве источника и приемника могут использоваться только сокращенные ссылки.

Аргументы программы копируются в массив arg.

```
COPY arg$=A$;
```

Для перемещения можно использовать команду **MOVE**.

```
MOVE a1$=b1$;
```

Копировать и перемещать можно любые массивы.

Синхронизация ресурсов.

При выполнение нескольких заданий возникает необходимость доступа к общим ресурсам. Синхронизацию выполняют команды блокировки. Команды синхронизации сами по себе ничего не блокируют, это набор соглашений позволяющих разграничить доступ к общим ресурсам. Если команды блокировки не будут совместно использоваться в разных заданиях, то никакой синхронизации доступа не произойдет. Синхронизация построена на именах блокировок. Эти имена локализованы внутри приложения и одинаковы во всех заданиях. В блокировках принята концепция многих читателей и одновременно только одного писателя. Соответственно есть блокировки по чтению и есть блокировки по записи.

Команда **LockR** блокирует имена по чтению. Имена перечисленные в ее аргументах будут блокированы по чтению. Любое количество заданий может заблокировать эти же имена, но задание попытавшееся заблокировать любое из этих имен будет ожидать разблокирования всех этих имен. После захвата имени по записи ни одна команда блокировки по чтению не сможет выполнится пока блокировка по записи не освободит имя. При невозможности выполнить блокировку команда будет ожидать освобождения имени.

Например:

LockR name1,name2,name3;

Эти имена будут блокированы по чтению. Другое задание так же может выполнить блокировку этих имен не дожидаясь их разблокирования по чтению

Команда **LockW** блокирует имена по записи. Имена перечисленные в ее аргументах будут блокированы по записи. Если имена перечисленные в аргументах уже заблокированы любой командой блокировки, то эта команда будет ожидать освобождения этих имен.

Например:

LockW name1,name2,name3;

Эти имена будут блокированы по записи.

Команда **LockUn** снимает блокировку этого имени. Если имя заблокировано по чтению несколько раз, то и снимать блокировку необходимо столько же раз.

Например:

LockUn name1,name2,name3;

В стандартных функциях есть аналоги этих команд со временем ожидания.

Сокращенная форма команды SET.

Команда **SET** имеет сокращенную форму. В этом случае левая часть равенства отсутствует, ее роль выполняет последняя упомянутая в выражении переменная.

Например:

SET $1=2,$1+3;

Переменная $1 станет равной 5.

Если в выражении несколько переменных, то результат будет присвоен последней переменной.

Например:

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

Переменная $3 станет равной 1+2+3=6. Хотя такую форму целесообразнее использовать только в очень простых случаях, на подобии первого примера. Второй пример приведен только в качестве иллюстрации возможностей этой формы команды **SET.**

Блочные команды.

Блочные команды образуют блок команд и служат одновременно заголовком блока. Каждой блочной команде должна обязательно соответствовать своя команда **END**, даже если в блоке всего одна команда.

Команда IF.

Команда **IF** образует блок, который выполняется в случае истинности условия выполнения этой команды. Эта команда аргументов не имеет. Внутри этого блока могут находиться команды **ELSE**. Команда **ELSE** не имеет аргументов. Вне блока **IF** эти команды смысла не имеют и применяться не могут. В случае присутствия команд **ELSE** в блоке **IF** при выполнении условия команды **IF**, выполняются только команды находящиеся за командой **IF** до следующей команды **ELSE**. Команда **ELSE** может содержать условие выполнения, тогда в случае истины в этом условии будут выполнены только команды расположенные после этой команды до следующей команды **ELSE** или команды **END**. В блоке **IF** может находиться только одна команда **ELSE** без условия ее выполнения и она должна быть последней среди команд **ELSE**.

Например:

```
IF?[6]<0;
 SET y[1]=1;
ELSE?[6]<5;
 SET y[1]=2;
ELSE?[6]<10;
 SET y[1]=3;
ELSE; SET y[1]=4;
END;
```

Условие выполнения этой команды может отсутствовать, тогда этот блок будет выполняться в любом случае. Хотя трудно представить для чего бы это могло пригодится.

Команда CASE.

Семантика этой команды несколько отличается от семантики остальных команд MSH. Условие выполнения команды в ней таковым не является. В условии выполнения команды **CASE** выражение должно вычислить метку на которую будет передано управление. Эта команда не имеет аргументов.

Каждая метка в этой команде образует блок от этой метки до следующей. При передачи управления на метку, выполняются только команды до следующей метки, а затем происходит выход за текущий блок **CASE**. Эта команда ближе соответствует нотации языка Паскаль, чем команде switch языка Си. В случае если в результате вычисления имени метки будет найдено имя отсутствующее в текущем блоке **CASE** будут выполняться команды находящиеся после команды **CASE** до первой метки.

```
CASE?L_$J; //вычисляется метка
 SET x[1]=1; //если метка не обнаружена то выполняются команды этого блока
 SET a[2]=x[1]+1;
```

L1: **SET x[1]=2;** //блок команд метки L1

SET a[2]=x[1]+2;

L2: **SET x[1]=3;** //блок команд метки L2

SET a[2]=x[1]+3;

END;

Метки в этой команде неявно образуют внутренний блок команд. После выполнения такого блока команд управление передается вне блока команды **CASE**.

Команда WHILE.

Команда **WHILE** используется для организации цикла. Условие выполнения этой команды задает условие продолжения цикла. Пока условие выполнения команды не равно 0, блок образованный этой командой будет выполняться. Завершается блок командой **END**. Команда **END** такого блока имеет особенность. Она может иметь условие завершения блока. Если условие не равно 0, то блок будет завершен. Причем эти условия могут присутствовать как в команде **WHILE**, так и в команде **END**.

Например:

WHILE?x[7]>0; //условие продолжения цикла

SET y[2]=x[7]+2;

BREAK?y[2]<0;

SET x[7]=x[7]+1;

END?x[7]>20; //условие завершения цикла

Внутри блока цикл может быть прерван командой **BREAK**.

Блочные итераторы цикла.

Блочные итераторы цикла оптимизированы по доступу к узлам дерева. В них используется внутренняя ссылка позволяющая оптимизировать доступ к узлам обхода. Это накладывает ограничение на использование блочных итераторов. Внутри блока нельзя менять структуру дерева. Писать в это дерево нельзя. В командах блочных итераторов циклов 2 аргумента.

Первый аргумент обязательный, ссылка на узел потомки которого будут обходится. Опорный индекс. Может использоваться сокращенная ссылка. Второй аргумент это ссылка на узел в котором будет сохранен индекс потомка. Этот аргумент необязателен. Внутри блока 2й аргумент меняться не должен. Выражения типа $$2 или [[3]] недопустимы в случае если $2 или [3] меняются внутри блока итератора. Изменение этих переменных не будет учтено. Доступ к индексу потомка можно получить через системную переменную **%queryKey**, а данные узла из свойства **%queryData**. Если потомков необходимо обходить не с начала, то 2й аргумент обязателен и в него надо поместить индекс узла за которым начнется обход потомков. Если 2й аргумент есть, но обход надо вести с начала, то перед циклом необходимо удалить эту переменную командой **KILL**.

Команды могут иметь условие выполнения блока. Это условие проверяется только один раз при входе в блок.

Обходить можно не только узлы дерева, но массивы. В этом случае во второй аргумент попадает порядковый номер поля массива. В качестве первого аргумента может использоваться только сокращенная ссылка.

Команда NEXT.

Команда **NEXT** итератор по непосредственным потомкам узла дерева. Потомки обходятся в прямом направлении от минимального индекса к максимальному.

Например:

```
NEXT us[4,5]; //2 аргумент не указан, индекс берется
 // из системной переменной %queryKey
 SET $1=%queryKey,$2=%queryData;
 END?$1>1000; //условие завершения цикла
```

Индекс узла сразу помещается в данные

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

В качестве опорного узла применена сокращенная ссылка. В этом случае обходится первый уровень дерева.

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

Обход производится после индекса 3.

```
SET $1=3;
NEXT us[4,5],$1;
 SET $2=%queryData;
 END;
```

При обходе массива, обходятся по порядку все поля даже те в которых данные не определены.

Например:

```
KILL $1;
```

NEXT us$,$1;

SET $2=%queryData;

END;

Внутри блока цикл может быть прерван командой **BREAK**.

Команда BACK .

Команда **BACK** отличается от команды **NEXT** только направлением обхода от последней вершины к первой.

Например:

KILL $1;

BACK us[4,5],$1;

SET $2=%queryData;

END;

Команда QUERY.

Команда **QUERY** обходит всех потомков узла слева направо и сверху вниз на полную глубину в прямом направлении. Во 2 аргумент помещается весь дополнительный индекс. Если в этом индексе больше 1 поля, то во 2 аргумент помещается список.

В остальном эта команда аналогична команде **NEXT**.

Обход массива команда **QUERY** осуществляет только в прямом направлении слева направо. Например:

KILL $1;

QUERY us[4,5],$1;

SET $2=%queryData;

END;

Но обходит эта команда только вершины имеющие значения.

Не блочные команды Команды обхода дерева данных.

В этих командах оба аргумента команды обязательны. Во втором аргументе сохраняется индекс, после которого следующая команда найдет следующую вершину. В этих командах не сохраняются внутренние ссылки и поэтому отсутствует ограничение на корректировку дерево которое обходится. По этой же причине время доступа к вершинам может быть намного больше. Массивы так же можно обходить этими командами.

Команда NEXT1.

Команда **NEXT1** дает следующую вершину на том же уровне под опорным узлом.
Данные узла доступны в системной переменной %queryData.
Например:

SET $1=2;
NEXT1 us[1,4],$1; //даст узел находящийся на 3 уровне после узла us[1,4,2]

Команда BACK1.

Команда **BACK1** дает предыдущую вершину на том же уровне под опорным узлом.
В остальном она аналогична команде **NEXT1**.

Команда QUERY1.

Команда **QUERY1** дает следующую вершину ветви дерева при обходе всего узла сверху вниз и слева направо. Во 2 аргумент помещается весь дополнительный индекс. Если в этом индексе больше 1 поля, то во 2 аргумент помещается список.
В остальном она аналогична команде **NEXT1**.

Управление выполнением программ.

Метка модуля может являться точкой вызова программы, функции, точкой вызова нового задания, свойством объекта, методом объекта и меткой, в зависимости от обращения к этой метке.

Причем к метке можно обращаться по разному в разных частях модуля.

Например:

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

Do LB(78); //Обращение к программе. Возвращаемое значение игнорируется.

Set Val[7]=8*LB(6); //Обращение как к функции возвращаемое значение используется.

JOB LB(17, «yt»); //вызов нового задания

Set Val[9]=Mod.LB(3); //Обращение к методу класса Mod- имя модуля в данном контексте трактуется как имя класса объекта.

Set Val[15]=Obj[1,2,A].LB; //Обращение к свойству LB объекта Obj[1,2,A],

Go LB; //Переход на метку LB

В приведенных примерах кроме 4 и 5 обращение осуществляется внутри текущего модуля. Вызов программ и функций может осуществляться к любому доступному модулю. Кстати к команде **Go** это тоже относится.

Например в модуле Mod есть метка Lb. Тогда обращение к ней будет иметь вид:

Do Mod.Lb(78);

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

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

JOB Mod.Lb(78);

Go Mod.Lb;

В общем случае метка используется как программа или функция. Если модуль используется как класс объекта, метка является свойством или методом этого объекта. Свойства объекта и его методы являются другой формой обращения к программам и функциям, поэтому все

сказанное о программах и функциях в равной мере относится и к свойствам и методам. Есть не значительные нюансы при обращении к объектам только в части наследования. Более детально этот вопрос будет изложен далее.

В общем случае имя модуля и имя метки в обращении к ней являются выражениями и могут вычисляться в момент обращения.

Например:

Set $1=“Md“,$2=“Lb“;

Do $1.$2;

Программа завершается командой Return; Модуль может быть завершен командой End;

Передача параметров.

В языке MSH передача параметров осуществляется только по значению. И вообще никаких ссылок и указателей в терминах языка C++ и связанных с их использованием колоссальных проблем здесь в принципе не существует. Программа всегда принимает произвольное число параметров в массиве A$. Их количество можно узнать обратившись к 0 элементу массива A$0. За смысл этих параметров отвечает программист.

Например:

У нас есть программа Lb. К ней возможны обращения:

DO Lb(1,2,7);

DO Lb(25,7);

DO Lb();

JOB Lb(8);

SET [7,8]=Lb(187,«Privet»);

Не переданные параметры в программе не определены.

Передача параметров по значению не мешает передавать внутрь программы имена переменных и там их использовать. Например:

Set us[5]=48;

Do Lb(«us»);

…

Return;

Lb: **Set par[8,1,9]=A$1[5];** // A$1 — первый переданный параметр содержит имя

//переменной: us.

//[5]- обращение к вершине этого дерева

// в результате в узел par[8,1,9] будет помещено значение узла us[5] = 48

Return;

В результате манипулировать именем переменных можно произвольным образом.

При переходе на метку командой **GO** массив параметров не меняется. В MSH есть форма команды **GO** с передачей новых параметров. В этом случае текущий массив параметров заменяется новым указанным в вызове команды **GO**.

Например:

GO Lb(1,2,7);

Обработка событий.

Обработка событий мощный механизм программирования, который как правило не включен в языки высокого уровня. Но его присутствие в библиотеках и операционной системе говорит о его важности. Библиотеки визуальных компонентов построены на этом механизме. Наличие обработки событий в языке расширяет возможности языка. Открывает новый стиль программирования основанный на обработке событий. Кстати такой уважаемый язык программирования как Ассемблер, такие возможности имеет. Поэтому этот механизм был добавлен в язык MSH.

Основой этого механизма является событие. Оно локализовано внутри приложения. Соответственно эти имена должны быть уникальны во всем приложении. Если имя события будет совпадать с именем объявленным в команде **CONSTANT**, то имя события будет заменено на значение объявленной константы. Будьте внимательны. При именовании событий желательно придерживаться какой либо стратегии имен. Например именам событий присваивать имена начинающиеся с evu. Длинна имени с учетом кодировки UTF8 не должно превышать 18 байт. Это ограничение связано только с текущей реализацией языка.

Событие созданное в одном задании видно и может быть обработано в любых заданиях. Обработчиков события может быть сколько угодно и находиться они могут в разных заданиях. После возникновения события проверяется есть ли обработчики этого события, если обработчиков нет, то событие удаляется. Если обработчики есть то они выполняются последовательно. События могут быть системными и пользовательскими. Системные события порождает система. Пользовательское событие создает команда **EVENTTRAP**. Событие передает обработчикам параметры, как при вызове программы. После обработки обработчики событий не удаляются. Для удаления события используется команда **EVENTDELETE**. Событие может быть обработано командами **EVENTCALL** и **EVENTWAIT**.

Эти команды позволяют организовать асинхронное выполнение программ.

Команда EVENTTRAP.

Команда создает пользовательское событие. Формат команды напоминает вызов программы, только вместо имени программы используется имя события. Имена событий локальны внутри приложения, поэтому они видны во всех заданиях.
Командой будет создано событие с указанным именем. Обработчикам событий будут переданы параметры перечисленные в аргументах команды.
Например:

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

Генерируется 2 события evuBoxData и evuKorXY в качестве параметров передаются переменные **us[7], $4, us[X], us[Y]** и строковая константа sh.
Если в данный момент нет обработчиков этого события, то событие не создается. Проверить в реализации??

Команда EVENTDELETE.

Команда удаляет обработчиков событий, перечисленных в аргументах программы.
Например:

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

События будут удалены в порядке следования в команде.

Команда EVENTCALL.

Команда назначает обработчик события. Обработчиком является программа. Эта программа будет вызвана асинхронно от выполняемого задания и ей будут переданы параметры. После выполнения программы обработчика управление будет возвращено основному заданию в место прерывания.

Например:

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

Программа обработчик будет вызвана в новом контексте программы. Контекст задания будет соответствовать контексту того задания в котором выполнялась команда **EVENTTRAP**.

Команда EVENTWAIT.

Команда ожидает возникновения событий. Эта команда заблокирует текущее задание до момента возникновения событий перечисленных в ее аргументах. Все события перечисленные в ее аргументах должны возникнуть для продолжения выполнения текущего потока. Текущие параметры программы заменяются переданными в команде создания события.

Например:

EVENTWAIT evuBoxData,evuKorXY;

<Нужна функция аналогичная этой команде со временем ожидания.>

Векторы.

Векторы тесно связаны с текущей реализацией. Хранят они знаковые и беззнаковые целые числа. Размерность этих векторов зависит от разрядности компоненты вектора и она фиксирована.

64 битные вектора.

Размерность такого вектора 2. Компонентами могут быть целые или целые без знака 64 битные числа. Хранятся такие числа в любых переменных.

Обращение к знаковым целым компонентам вектора.

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

%v64 — обращение к целой знаковой компоненте вектора.

Обращение к беззнаковым целым компонентам вектора.

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

%vu64 — обращение к целой беззнаковой компоненте вектора.

32 битные вектора.

Размерность такого вектора 5. Компонентами могут быть целые или целые без знака 32 битные числа. Хранятся такие числа в любых переменных.

Обращение к знаковым целым компонентам вектора.

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

%v32 — обращение к целой знаковой компоненте вектора.

Обращение к беззнаковым целым компонентам вектора.

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

%vu32 — обращение к целой беззнаковой компоненте вектора.

16 битные вектора.

Размерность такого вектора 11. Компонентами могут быть целые или целые без знака 16 битные числа. Хранятся такие числа в любых переменных.

Обращение к знаковым целым компонентам вектора.

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

%v16 — обращение к целой знаковой компоненте вектора.

Обращение к беззнаковым целым компонентам вектора.

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

%vu16 — обращение к целой беззнаковой компоненте вектора.

8 битные вектора.

Размерность такого вектора 22. Компонентами могут быть целые или целые без знака 8 битные числа. Хранятся такие числа в любых переменных.

Обращение к знаковым целым компонентам вектора.

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

%v8 — обращение к целой знаковой компоненте вектора.

Обращение к беззнаковым целым компонентам вектора.

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

%vu8 — обращение к целой беззнаковой компоненте вектора.

Операции.

Операции в MSH играют особую роль. Именно они управляют преобразованиями типов данных. В зависимости от операции операнды приводятся к нужному типу данных. Тип результата данных однозначно соответствует типу операции. Строковые операции и числовые не накладываются друг на друга, как операция «+» в Си подобных языках. Тип операции в MSH не зависит от типа операндов, все с точностью до наоборот. В языке MSH нет приоритетов операций, это историческое наследие MUMPS.

Например:

SET $1=2+3*4;

$1 будет равняться 20, а не 14.

Для того чтобы результат был 14 применяются скобки.

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

В качестве операции соединения строк используется символ «_».

Отсутствие приоритетов операций необычно, но довольно удобно. Ссылка на естественность приоритетов, когда операций больше чем сложить умножить становится очень сомнительной. Усвоив один раз, что приоритетов нет, нет необходимости мучительно вспоминать их приоритеты, а если что то и лезть в документацию. И вообще это дело привычки.

Объекты.

Наличие объектов в современных языках программирования является хорошим тоном. В общепринятом случае объекты состоят из 2 частей. Части декларативного описания и части реализации. В MUMPS системах переменные не имеют декларативной части описания типа. Классы в принципе являются пользовательскими типами данных. Чтобы не нарушать принципы MUMPS в MSH отсутствует декларативная часть описания класса. И как оказалось без нее прекрасно можно обойтись. От класса осталась только часть реализации. Реализация класса прекрасно может быть представлена стандартным модулем. Только для класса пришлось ввести дополнительные соглашения связанные с описанием свойств объекта. Объекты могут находиться только в дереве. Разместить объект в массиве не получится так как негде хранить свойства объекта. Хотя если свойств у объекта нет, то можно попробовать. Хотя какой это объект.

Для описания публичного свойства понадобится функция возвращающая значение этого свойства. Она должна иметь имя свойства и находиться в модуле с именем класса. Программа записи в свойство. Эта программа имеет имя свойства с префиксом «.». И программа удаления свойства. Эта программа имеет имя свойства с префиксом «..».

Имя функции в модуле может соответствовать публичному свойству по чтению. При этом возвращаемое значение такой функции передается вызвавшей программе в качестве значения публичного свойства.

Конструктором класса является метод данных **%objNew**. Если при создании объекта необходимо определить какие либо свойства или получить ресурсы, то можно использовать любую программу модуля класса (метод класса). Но желательно придерживаться какой либо

стратегии наименования конструкторов класса. Например имя конструктора должно совпадать с именем класса.

Доступ к защищенные свойствам класса осуществляется через системное свойство **%this**.

В качестве деструктора служит удаление объекта командой **KILLD**. Если необходимо освободить какие либо рессурсы или выполнить дополнительные манипуляции, то это можно выполнить любой программой этого класса (метод класса). Как и в случае конструктора при наименовании деструктора желательно придерживаться какой либо стратегии наименования.

Например:

```
//Класс  Person
        //Свойство Age
        //чтение публичного свойства Age
Age:    RETURN [%this,Age];
        //запись публичного свойства Age
.Age:   SET [%this,Age]=A$1;
        RETURN;
        END;
```

Обращение из программ к объекту и его публичным свойствам будет выглядеть следующим образом.

```
        //создание объекта Person
        SET us[1,2].%objNew=Person;
        //записать в свойство Age значение 50
        SET us[1,2].Age=50;
        //прочитать свойство Age
        SET u$1= us[1,2].Age+5;
        //удаление свойства Age
        KILL  us[1,2].Age;
```

Наследование объектов.

Классы MSH поддерживают множественное наследование. Команда **PARENT** задает всех предков данного класса. Причем порядок следования имен предков данного класса в аргументах команды определяет приоритет наследования. Чем позже упомянут класс, тем приоритет его ниже.

Например:

```
        PARENT USER,BOX;
```

Класс наследуется от предков USER и BOX. Приоритет предка USER выше. На что влияет приоритет. При обращении к объекту публичное свойство или метод класса будет искаться в самом классе, если они там обнаружены не будут, то их будут искать в предке с наивысшим приоритетом, а затем в предках этого класса и так далее по приоритетам.

Обмен с файлом.

В MSH обмен с файлами организован на самом примитивном уровне. Файлы в MSH играют вспомогательную роль. Обмен организован только с текстовыми файлами. Структура файла это текстовые поля разделенные заданным разделителем. Разделитель находится в системной переменной **%dlmIO**. По умолчанию эта переменная равна «,». Она доступна по чтению и записи. При записи в файл переменные преобразуются к строковому типу и записываются в файл через разделитель. При чтении из файла, переменные выбираются через разделитель и приводятся к нормализованному виду. Если поле представляет из себя запись числа, то в переменную помещается число. Обмен с файлом происходит через массив **B$**. При записи массив **B$** записывается в файл через разделитель. При чтении из файла поля выбираются в массив **B$**.

Команды обмена с файлом имеют в качестве аргументов путь к файлу.

Например:

Из массива **B$** данные записываются в файл. Файл открывается по записи. Данные в файле заменяются.

```
Write «txt/tt1.txt»;
```

Данные читаются в массив B$. Массив предварительно очищается.

```
Read «txt/tt1.txt»;
```

txt/tt1.txt — путь к файлу.

Заключение.

Данный документ не является заменой описанию языка MSH, а только дополняет его. Здесь рассмотрены не все возможности языка MSH, а только те на которые я хотел обратить ваше внимание.

Автор: Шарымов Михаил Алексеевич. Email : misha_shar53@mail.ru