Описание языка MSH

Оглавление

1. Введение

Основой для языка MSH послужил язык MUMPS. MUMPS разработан где то в 60-х 70-х годах прошлого столетия, как впрочем и многие другие современные языки. Он с самого начала разрабатывался как язык для создания больших информационных систем, работы с большими распределенными данными. В связи с чем имеет некоторую специфику. Программирование на этом языке несколько отличается от программирования на других языках. Несмотря на его малую распространенность у него существует довольно устойчивое сообщество в разных частях света программистов разрабатывающих на нем информационные системы. За время своего существования MUMPS мало изменился.

Некоторые элементы языка потеряли актуальность. И язык на данный момент несколько архаичен. Хотелось бы иметь более современный язык подобный MUMPS. С этой целью и создавался язык MSH. Некоторые концепции языка MSH отличаются или отсутствуют в языке MUMPS.

Цель

Цель: -Сократить время обслуживания цикла жизни программного обеспечения.

Сокращение времени разработки достигается следующими средствами:
-Язык должен быть синтаксически простым лаконичным и минимально возможным.
-Конструкции языка должны быть однозначными и не допускать необычного использования.
-Набор команд языка должен быть достаточным что бы поддерживать все современные модели программирования.
-Язык должен быть максимально гибким. Достигается отсутствием декларирования переменных.

Сокращение времени отладки достигается следующими средствами:
-Простотой и наглядностью программ.
-Минимально возможных ошибок в программе.
-Программы должны содержать минимум строк.

Сокращение времени сопровождения достигается следующими средствами:
-Надежностью языка.
-Язык должен иметь средства обеспечения целостности данных.
-Минимально возможных ошибок в программе.

Ни один из современных языков не соответствует поставленной цели. Сейчас имеется большой выбор различных языков каждый из которых содержит какие либо идеи. Но идей не так уж много. Попытаюсь самые ценные идеи использовать в языке.
Для достижения максимальной гибкости язык должен обладать определенными свойствами в частности он не должен быть декларативным. Все декларативные языки немедленно теряют гибкость. Программы на них сразу раздуваются в объеме. Сами деклараци переменных кроме помощи транслятору никакого другого смысла не имеют. Язык должен быть ориентирован на манипулирование данными. Языком наиболее полно отвечающим поставленной цели является язык программирования MUMPS. Его и берем за основу. Последним стандартом этого языка является стандарт 1995 года он и будет отправной точкой.

2. Лексика языка

Алфавит

Алфавитом является набор символов в кодировке UTF-8.

Комментарии

однострочный комментарий //
многострочный комментарий /* */

При описании языка необязательные параметры помещаются внутри скобок < >

3. Операции

Операндами могут быть любые переменные, константы и функции.

Унарные операции:

'-' унарный минус Признак отрицательного числа.

Результат : Арифметический тип.
'~' Логическая операция НЕ
Результат : целое число 0 или 1
Бинарные операции:
Операции могут быть 1-2 символьными.
Арифметические операции: Результат Арифметический тип
'+' плюс
'-' минус
'*' умножить
'**' возведение в степень
'/' деление
'\' деление нацело Результат целое число
'#' модуль числа Результат целое число
Логические операции: Результат целое число 0 или 1
'&' операция И
'|' операция ИЛИ
Операции сравнения: Результат целое число 0 или 1
'>' больше
'<' меньше
'==' равно
'~=' не равно
'~>' не больше
'~<' не меньше
'>=' больше или равно
'<=' меньше или равно
'<>' не равно
Строковые операции: Результат строка символов
'_' сцепление строк
'_>' строка следует
'_<' строка содержит

Побитовые операции над байтами :
'_&' операция И
'_|' операция ИЛИ
'_^' операция исключающее ИЛИ
'_~' операция НЕ

Операция выбора:
'?'-почти соответствует стандарту языка Си.
expYcl ? *expTrue* ! *expFalse*
expYcl-условие выбора,
expTrue-выражение присваиваемое при выполнении условия,
expFalse-выражение присваиваемое при Не выполнении условия.

Встроенные операции имеют вид операций с префиксом `
Arg1 `name *Arg2*

4. Константы

В MUMPS константы отсутствуют. В MSH они введены. Но из-за специфики языка существовать в виде неизменяемых переменных времени исполнения программы они не могут. Поэтому в MSH они введены как объекты времени компиляции. Что то типа define

языка Си в самом упрощенном виде. В момент трансляции языка MSH в Pi код виртуальной машины все константы в исходном тексте заменяются их значениями. То есть в Pi коде их уже нет. Константы могут использоваться как значения переменных, в качестве индекса или имени переменной.

Константы состоят из имени и значения. Имя есть идентификатор. Значение константы есть любое количество алфавитно цифровых символов на любых языках. Значение константы может быть в кавычках или в двойных кавычках. В этом случае внутри кавычек могут присутствовать любые символы.
Формат команды:

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

Справа от знака равенства может находиться только константа, выражение не допускается. Имя константы должно быть уникальным и с именами переменных не пересекаться.
Пример значений констант:
Constant cCnst1=123 ABC, **cCnst2** =15Нгш, **cCnst3** ='75&56+»()»', **cCnst4** = «wq^erНГ"»;
Вторая форма констант имеет вид:
Constant *nameMod <,nameMod>*;
nameMod- имя модуля из которого будут импортированы константы. Это позволит иметь библиотеки констант.
Пример:
Constant *modLibSys ,modLibUser*;
Где *modLibSys* и *modLibUser* – модули содержащие константы. На момент трансляции они должны существовать.

5. Переменные

В MUMPS переменные хранятся в деревянной структуре, будь то глобали или локали. Правда у глобалей в MUMPS есть небольшое отличие — сокращенный синтаксис. В MSH нет сокращенного синтаксиса. Хранение данных в виде только дерева позволяет унифицировать обращение к данным. Можно писать программы обработки любого дерева данных. Но в программах часто приходится работать с различными промежуточными значениями и хранить их в дереве не эффективно. Доступ к дереву всегда влечет дополнительные расходы, как бы мы этот доступ не оптимизировали. В MSH данные могут храниться как в деревянной структуре, так и в одномерном массиве. В одномерном массиве лучше хранить промежуточные данные, хотя можно хранить и любые данные. Доступ к массиву значительно быстрее, чем к дереву. Дерево хранит так называемые разряженные данные. То есть в дереве хранятся только существующие вершины. Например имеем дерево [1] [5] [78]. Только эти вершины и будут хранится в дереве. Массив хранит все промежуточные значения. Например мы создали 78 элемент в массиве $78. Будут созданы все элементы в этом массиве вплоть до 78.

Видимость переменных в MUMPS имеет свои особенности. То что в MUMPS называют глобалями в других языках это обращение к базам данных. А вот локали являются аналогом переменных в других языках. Ну с глобалями все ясно. Это внешние данные и они доступны как из любого задания данного приложения, так и из других приложений. Хранятся они в виде файлов и как правило несколько глобалей в одном файле. В MSH в общем то все так же за исключением одного момента. Каждая глобаль хранится в отдельном файле. А если точнее то в нескольких файлах одного каталога. Этот момент существенен с технологической точки зрения построения информационной системы. И еще одно замечание. Глобали в MSH синхронизированы по обращению. Это значит, что к ним можно обращаться одновременно из

разных заданий одного приложения без дополнительной синхронизации.

Теперь детальней разберемся с локалями. В общем случае локаль в MUMPS является глобальной переменной верхнего уровня в смысле языка Си. Аналога переменным из кучи в языке Си, в MUMPS нет. А вот аналог автоматическим переменным языка Си есть. Это локали перечисленные (или не перечисленные) ,в зависимости от формы команды, в команде New. Область видимости таких локалей от команды New до команды завершения блок Quit. Оригинальное конечно решение обусловленное природой MUMPS и отсутствием в нем деклараций переменных. По моему мнению глобальная область видимости локалей по умолчанию спорное решение, хотя и не фатальное. В MUMPS в основном работа напрямую с глобалями, локальных переменных не так уж много и конфликты имен не часты, хотя и случаются. Но все же это не удобно при программировании. В MSH принят другой подход. Команды New здесь нет. А локализация переменных выполнена по префиксу. Внутри блока Do локализована только одна переменная без имени. Переменные имеющие префикс % локализованы внутри приложения. Переменные не имеющие этих префиксов локализованы внутри задания. Это касается одинаково как деревьев, так и массивов.

В языке отсутствуют объявления переменных. Переменная возникает в момент когда ей присваивается значение. При обращении к неопределенной переменной возвращается пустая строка.
Переменные могут находиться как в оперативной памяти так и во внешней памяти.
Переменные находящиеся во внешней памяти называются глобали и имеют префикс ^.
Например: ^abc[15,gh8,42] abc- имя глобали.

Переменные находящиеся в оперативной памяти называются локали.
Локали разделяются по области видимости.
1. Переменная блока Do. Эта переменная видима только внутри блока Do. Эта переменная не имеет имени. Например [1,2] или $4.
2. Переменные приложения. Они являются общими для всех программ в приложении. Эти переменные имеют префикс %. Например %a1[1,2] или %b$4.
3. Остальные переменные . Они видны только внутри данного задания. Например a1[1,2] или b$4.

Переменные могут иметь 2 структуры хранения.

Структура дерева.
Такие переменные выглядят в виде произвольной индексной структуры заключенной в квадратные скобки. Индексы разделены запятыми. Любой индекс может иметь любой тип. Число, строку и др. Например : ldb[abc,125,5,9]
Индексы в переменных могут быть произвольными выражениями.
В имени может быть использована предопределенная константа %this. В этом случае идет обращение к закрытым свойствам объекта. Такое обращение возможно только внутри вызова метода или свойства объекта.
Например:
[%this].Age

Структура сплошного массива.
Эта структура имеет один целочисленный индекс. Все элементы массива расположены подряд. Эта структура применяется для быстрого доступа к данным.
Индекс массива следует за знаком $
Например : db$25
Индексация массива начинается с 1. Нулевой элемент массива является служебным и содержит индекс последнего элемента. Итераторы игнорируют этот элемент.

В программе MSH возможна как полная ссылка как на конкретные данные дерева или массива, так и сокращенная ссылка на дерево или массив целиком. Полная ссылка содержит имя переменной и полный индекс.
Например:
ab[a,gt,125] или ab$25.
сокращенная ссылка содержит имя и пустой индекс.
Например:
ab[] или ab$
Конструкции где допустима сокращенная ссылка оговариваются особо. Возможна так же ссылка на дерево и массив одновременно, тогда указывается только имя.
Например: ab
Где возможно такое обращение оговаривается в конкретной команде особо.

6. Псевдомассивы.

В них передается служебная информация. Обращение к ним соответствует обращению к массиву.
Перечень псевдомассивов:
префикс A$ - значения переданных в функцию аргументов, открыт по записи,
префикс B$ - буфер ввода вывода, открыт по записи,
префикс J$ стек вызовов заданий для текущего блока Do, открыт только по чтению.

7. Выражения

Выражения это комбинация операндов и операций. Операнды могут являться выражениями, тогда они заключены в скобки. Типы переменных произвольны. Результат операции однозначно определяется операцией. Приоритеты операций отсутствуют.

8. Команды

Аргументы в различных командах могут быть разными. Аргумент может состоять из подполей разделенных символом =
Для дальнейших объяснений примем следующие соглашения:
exp — префикс обозначающий выражение
expYcl — выражение задающее условие выполнения команды
ref — префикс обозначающий ссылку на переменную.
refTree — ссылка на переменную структуры дерева.
Prog — ссылка на программу в виде *<expMod>.expLabel<(expArg <,expArg>)>*
expMod — выражение значение которого есть имя модуля
expLabel — выражение значение которого есть метка в модуле
expArg - выражение значение которого является аргументом. Аргумент передается по значению.

Интервалы времени задаются в микросекундах 10e-6 сек.

Декларативные команды:
Parent , Constant.
Декларативные команды имеют формат:
CMD *Arg <,Arg>*;
CMD — код команды.
Arg — аргумент команды.
Декларативные команды выполняются в момент трансляции по мере появления в исходном тексте программы. Их выполнение не зависит от хода выполнения программы. Условие

выполнения команды к ним не применимо.

Команда **Parent**.

Аргументами команды являются предки данного модуля. Команда используется для организации наследования.
Parent *expParent <,expParent>*;
expParent- константное имя родителя для данного модуля. Как видно из синтаксиса имеет место множественное наследование. Приоритет выбора родителя определяется порядком следования в списке наследования. Чем раньше встречается родитель тем его приоритет выше.

Команда **Constant** (**Const**).

Команда **Constant** имеет 2 формы.

1 форма :

Constant *nameConst=valConst <,nameConst=valConst>*;
nameConst- имя константы есть идентификатор и его длинна не должна превышать 18 байт,
valConst- значение константы. Может состоять из алфавитно цифровых символов или любых символов заключенных в кавычки или двойные кавычки.

2 форма :

Constant *nameMod <,nameMod>*;
nameMod- имя модуля из которого импортировать раздел констант.

Команды выполнения:
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.

Команды состоят из :
- кода команды,
- необязательного условия выполнения команды,
- обязательного пробела,
- необязательных аргументов перечисленных через запятую.

Конец команды отмечается точкой с запятой. Код команды не зависит от регистра и может быть сокращен. Полный код команды может быть расширен любым количеством латинских символов или цифр. Например команда End может быть записана как EndIf. Это же касается и других команд.
Условие выполнения команды отделяется от кода команды знаком ? Если результат вычисления условия выполнения равен 0 то команда не выполняется. Если результат вычисления условия выполнения не равен 0 то команда выполняется.
CMD*<?expYcl> <Arg1 <,ArgN>>*;

Команда вычисления выражения **Set**.

Команда **Set** имеет 2 формы.

Полная команда **Set**:

Set (S) - имеет вид:
Set*<?expYcl> ref=exp <,ref=exp>*;
S*<?expYcl> ref=exp <,ref=exp>*;
переменой *ref* присваивается значение выражения *exp*.

Например: Set ab[1,2]=$1+78*7;
Сокращенная команда **Set** не имеет правой части

Set<?*expYcl*> *exp* <,*exp*>;
S<?*expYcl*> *exp* <,*exp*>;
Внутри выражения exp должно присутствовать обращение к переменным, к последней из которых и присваивается значение выражения *exp*.
Например: **Set** 9-[2]-ab[1,2]+78*7;
Переменной ab[1,2] будет присвоено значение выражения.
Например: **Set** $1=10; **Set** $1+2;
Переменной $1 будет присвоено значение 12.

Команда удаления данных **Kill**.

Команда **KiLL** (**K**).
Kill<*?expYcl*> *ref* <,*ref*>;
K<*?expYcl*> *ref* <,*ref*>;
Команда удаляет потомков узла *ref* и сам узел и свойства объекта. В этой команде может использоваться сокращенная ссылка.
Имеется еще модификации этой команды.
Удаляет данные в узле *ref*.
KillD<*?expYcl*> *ref* <,*ref*>; **KD**<*?expYcl*> *ref* <,*ref*>;
Удаляет потомков узла *ref*.
KillN<*?expYcl*> *ref* <,*ref*>; **KN**<*?expYcl*> *ref* <,*ref*>;
Удаляет свойства объекта узла *ref*.
KillP<*?expYcl*> *ref* <,*ref*>; **KP**<*?expYcl*> *ref* <,*ref*>;
Возможны комбинации постфиксов например
Удаляет данные и свойства объекта узла *ref*.
KillDP<*?expYcl*> *ref* <,*ref*>; **KP**<*?expYcl*> *ref* <,*ref*>;
Удаляет данные и потомков узла *ref*.
KillDN<*?expYcl*> *ref* <,*ref*>; **KN**<*?expYcl*> *ref* <,*ref*>;
Если необходимо удалить только дерево, то используется команда.
Kill ^gb[]
А если только массив то
Kill ^gb$
В этой команде так же допустимо указать только имя. В этом случае будет удалено как дерево, так и массив с этим именем.
Например:
Kill ^gb
Очевидно, что такая команда не может использоваться для локальных переменных блока Do. По причине отсутствия имени у такой переменной.

Команды копирования данных (**Copy** и **Move**).

Узел данных *refFrom* сливается с узлом *refTo*. В этой команде могут быть использованы сокращенные ссылки.
Copy<*?expYcl*> *refTo=refFrom* <,*refTo=refFrom*>;
Узел данных *refFrom* перемещается в узел *refTo*
Move<*?expYcl*> *refTo=refFrom* <,*refTo=refFrom*>;

Команды блокировки задания (**Lock**).

LockW (LW).
LockW<*?expYcl> expID <,expID>*;
LW<*?expYcl> expID <,expID>*;
Устанавливает блокировку по записи на идентификатор заданный выражением *expID*.

LockR (LR).
LockR<*?expYcl> expID <,expID>*;
LR<*?expYcl> expID<,expID>*;
Устанавливает блокировку по чтению на идентификатор заданный выражением *expID*.

LockUn (LU).
LockUn<*?expYcl> expID <,expID>*;
LU<*?expYcl> expID <,expID>*;
Снимает блокировку с идентификатора заданного выражением *expID*.

Команды изменения хода выполнения программы.

Команда **TimeOut.**

Выполняет задержку выполнения текущей программы.
TimeOut<*?expYcl> expTime<,expTime>*;
expTime — выражение задающее задержку вычислений в текущей программе.
Интервалы времени задаются в микросекундах 10е-6 сек.

Команда **GOTO (G).**

Выполняет переход в новую точку выполнения программы.
GOTO<*?expYcl> expMod.expLabel< (<arg <,arg> >) >*;
G<*?expYcl> expMod.expLabel< (<arg <,arg> >) >*;
переход на метку в том же модуле.
GOTO<*?expYcl> expLabel< (<arg <,arg> >) >*;
G<*?expYcl> expLabel< (<arg <,arg> >) >*;
expMod — результат вычисления этого выражения является именем модуля.
expLabel — результат вычисления этого выражения является точкой продолжения выполнения программы.
При переходе по команде **GOTO** возможна замена аргументов на указанные в скобках.

Команда **Do (D)**.

Команда выполняет вызов подпрограммы без локализации переменных. Внутри блока локализована только переменная без имени.
Do<*?expYcl> <expMod.>expLabel< (<arg <,arg> >) >< ,<expMod.>expLabel< (<arg <,arg> >) > >*;

Команда **Job (J)**.

Команда выполняет программу в новом задании. Для каждого такого вызова создается новый блок Do. По завершению задания формируются код завершения. В ходе выполнения задания доступен код состояния задания.

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

<,arg>>) > ;

Команда **Return (Re).**

Команда завершает выполнение текущей программы и возвращает значение в вызывающую программу.
Return*<?expYcl> expRet*;
Re*<?expYcl> expRet*;
expRet- выражение вычисляется и возвращается вызывающей программе.

Команда **Break (*Br*).**

Выход из команд цикла может осуществляться в любой точке цикла по команде **Break (Br)**. Если команда **Break** встретилась внутри команды **Case** или **If** или **Else** , то выход происходит за пределы всех вложений этих команд.
Break*<?expYclFor>; Br<?expYclFor>;*

Команда **Xecute(X)**.

Команда вычисляет аргумент интерпретирует его как команду MSH и выполняет его.

Xecute*<?expYcl> exp<,exp>*;
exp-выражение вычисляется, интерпретируется как команда MSH и выполняется.

Блочные Команды.

Программа может содержать блоки. Блок начинается блочной командой и заканчивается командой **END**;

Команды **If (I)** и **Else(E)** .

Если условие выполнения этой команды не равно нулю то выполняются команды идущие за командой If вплоть до команды Else или End. Внутри блока If End могут встречаться команды Else.

Команда Else без условия выполнения может быть только одна и должна быть последней в цепочке команд Else. Если команда Else имеет условие выполнения и это условие не равно нулю то выполняются команды следующие за ней до команд Else или End.
Эти команды не имеют аргументов.
If<*?expYcl>;*
I*<?expYcl>*;
любые команды
Else*<?expYcl;>*
E*<?expYcl>*;
любые команды
END;
Команда Else в формате с условием выполнения фактически является командой ElseIf, а без условия выполнения является командой Else.
В общем виде блок If может выглядеть так :
If*?expYcl*;
comand...
Else*?expYcl;*
comand...
Else;
comand...

End;

Команда выбора **Case**.

Команда обеспечивает выполнение участка кода в зависимости от заданного условия *expYcl* .Условие в команде обязательно . Команда не имеет аргументов и используется в нотации Pascal.

Результат вычисления выражения *expYcl* должен давать строку.

Если эта строка равна одной из меток внутри блока **Case** то выполняются команды идущие после этой метки до следующей метки или команды **End**.

Если строка не равна ни одной метке то выполняются команды <*comandElse...*>. Команды **Case** могут быть вложенными, а также комбинироваться с любыми командами.

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

Например:

```
  Set $1=L1;
  Case?$1
   Set $2=’метка не совпала’;
L0:   Set $2=’метка L0’;
   Set $3=L0;

L1:   Set $2=’метка L1’;
   Set $3=L1;
L2:   Set $2=’метка L2’;
   Set $3=L2;
  End;
```

Команда **Transaction(Tr)**.

Команда **Transaction** является началом блока выполнения транзакций глобалей. Действие этой команды распространяется только на текущее задание. Если внутри этого блока встречается команда Job то на выполнение задания Job текущая транзакция распространяться не будет. Завершается блок командой End с одним аргументом. Если этот аргумент имеет значение =0 , то производится откат транзакции. Аргумент может отсутствовать тогда транзакция проводится. Если условие выполнения команды ложно то весь блок не выполняется.

Transaction<*?expYcl*>;
comand...
End <*exp*>;

Блочные Команды цикла.

Команда **While**.

Команда не содержит инициализации цикла. Инициализация цикла выполняется другими командами перед циклом. Если условие выполнения команды While становится равным 0 то цикл завершается. Если условие выполнения команды End становится не равным 0 то цикл завершается.

While<*?expYclWhile*>;

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

expYclWhile — условие продолжения цикла.
expYclEnd — условие завершения цикла.

Блочные итераторы цикла.

Эти команды используются для обхода данных. На каждом шаге становится доступной следующая вершина с данными.
*Замечание по реализации для команд **Next, Back** и **Query**. Эти замечания не касаются команд **Next1, Back1** и **Query1** :
Внутри блока итератора опорный индекс 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.
Во время обхода структура данных не должна меняться.

Команда **Next (N)**.

Обход дерева данных.

Команда обхода потомков узла на одном уровне с начала до конца.
Next<*?expYcl*> *refY*<,*refSaveInd*>;
comand...
End<*?expYclEnd*>;
N<*?expYcl*> *refY*<,*refSaveInd*>;
comand...
End<*?expYclEnd*>;

refY — ссылка на узел потомки которого будут обходится. Опорный индекс. Может использоваться сокращенная ссылка.
refSaveInd - ссылка на узел в который будут сохранен подиндекс потомка. Сохраняться будет только последний индекс потомка. Параметр необязательный. Данные узла доступны в системной переменной %queryData. Индекс узла доступен в переменной %queryKey. Если необходимо обойти свойства узла, то к индексу добавляется символ ':'. Если потомков надо обходить не с начала, то индекс после которого надо начать обход записываем в переменную *refSaveInd*, иначе эту переменную перед обходом надо удалить командой **Kill**.
<*?expYcl*>- условие продолжения цикла.
<*?expYclEnd*> - условие при котором цикл будет окончен. Если условие не равно 0, то цикл будет окончен.
Например **Next** [],$1
Например **Next** ^g[],$1
Например **Next** b1[4:],$1
Эта ссылка внутри итератора изменяться не должна. Любое изменение учтено не будет.
Например **Next** aa[4],$$1
В этом случае будет использовано только первое значение $1
Это замечание касается так же команд **Back** и **Query**.

Обход массива.

Команда обходит все элементы массива подряд. Не пропуская пустых значений.

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

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

Например **Next** $,$1
Например **Next** ^g$,$1
nameMacc — имя массива. Должна использоваться только сокращенная ссылка.
refSaveInd - ссылка на узел в который будут сохранен подиндекс потомка. Сохраняться будет только последний индекс потомка. Параметр необязательный. Данные узла доступны в системной переменной %queryData. Индекс узла доступен в переменной %queryKey. Если потомков надо обходить не с начала, то индекс после которого надо начать обход записываем в переменную *refSaveInd*, иначе эту переменную перед обходом надо удалить командой Kill.
<*?expYcl*>- условие продолжения цикла.
<*?expYclEnd*> - условие при котором цикл будет окончен. Если условие не равно 0, то цикл будет окончен.

Команда **Back (Ba)**.

Обход дерева данных.
Команда обхода потомков узла на одном уровне с конца до начала.
Back<*?expYcl*> *refY*<,*refSaveInd*>;
comand...
End<*?expYclEnd*>;
Ba<*?expYcl*> *refY*<,*refSaveInd*>;
comand...
End<*?expYclEnd*>;

refY — ссылка на узел потомки которого будут обходится. Опорный индекс. Может использоваться сокращенная ссылка.
refSaveInd - ссылка на узел в который будут сохранен подиндекс потомка. Сохраняться будет только последний индекс потомка. Параметр необязательный. Данные узла доступны в системной переменной %queryData. Индекс узла доступен в переменной %queryKey. Если необходимо обойти свойства узла, то к индексу добавляется символ ':'. Если потомков надо обходить не с начала, то индекс после которого надо начать обход записываем в переменную *refSaveInd*, иначе эту переменную перед обходом надо удалить командой **Kill**.
<*?expYcl*>- условие продолжения цикла.
<*?expYclEnd*> - условие при котором цикл будет окончен. Если условие не равно 0, то цикл будет окончен.

Например **Back** [],$1
Например **Back**^g[],$1
Например **Ba** b1[4:],$1
Эта ссылка внутри итератора изменяться не должна. Любое изменение учтено не будет.
Например **Back** aa[4],$$1
В этом случае будет использовано только первое значение $1
Это замечание касается так же команд **Back** и **Query**.

Обход массива.

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

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

Например **Back** $,$1
Например **Ba** ^g$,$1
nameMacc — имя массива. Должна использоваться только сокращенная ссылка.
refSaveInd - ссылка на узел в который будут сохранен подиндекс потомка. Сохраняться будет только последний индекс потомка. Параметр необязательный. Данные узла доступны в системной переменной %queryData. Индекс узла доступен в переменной %queryKey. Если потомков надо обходить не с начала, то индекс после которого надо начать обход записываем в переменную *refSaveInd*, иначе эту переменную перед обходом надо удалить командой **Kill**.
<*?expYcl*>- условие продолжения цикла.
<*?expYclEnd*> - условие при котором цикл будет окончен. Если условие не равно 0, то цикл будет окончен.

Команда **Query (Q)**.

Обход дерева данных.

Команда обхода всех потомков узла.
Команда **Query** обходит ветвь дерева сверху вниз и слева направо.
Query<*?expYcl*> *refY*<,*refSaveInd*>;
comand...
End<*?expYclEnd*>;
Q<*?expYcl*> *refY*<,*refSaveInd*>;
comand...
End<*?expYclEnd*>;

refY — ссылка на узел потомки которого будут обходится. Может использоваться сокращенная ссылка.
refSaveInd - ссылка на узел в который будут сохранен подиндекс потомка. Сохранен будет весь индекс в виде списка. Параметр необязательный. Если потомков надо обходить не с начала, то индекс после которого надо начать обход записываем в переменную *refSaveInd*, иначе эту переменную перед обходом надо удалить командой **Kill**.Данные узла доступны в системной переменной %queryData. Индекс узла доступен в переменной %queryKey.
<*?expYcl*>- условие продолжения цикла.
<*?expYclEnd*> - условие при котором цикл будет окончен. Если условие не равно 0, то цикл будет окончен.

Обход массива.

Команда аналогична команде **Next.** Но пустые значения пропускаются.

Не блочные команды Команды обхода дерева данных.

Команда **Next1 (N1)**.

Команда **Next1 (N1)** дает следующую вершину на том же уровне.
Next1<*?expYclEnd*> *refY,refSaveInd*;

refY — ссылка на узел потомки которого будут обходится. Может использоваться

сокращенная ссылка.
RefSaveInd - ссылка на узел в который будут сохранен подиндекс потомка. Сохраняться будет только последний индекс потомка. Параметр обязательный. Данные узла доступны в системной переменной %queryData.

Команда **Back1 (Ba1)**.

Команда **Back1 (Ba1)** дает предыдущую вершину на том же уровне.
Back1*<?expYclEnd> refY,refSaveInd*;

refY — ссылка на узел потомки которого будут обходится. Может использоваться сокращенная ссылка.
RefSaveInd - ссылка на узел в который будут сохранен подиндекс потомка. Сохраняться будет только последний индекс потомка. Параметр обязательный. Данные узла доступны в системной переменной %queryData.

Команда **Query1 (Q1)**.

Команда **Query1 (Q1)** дает следующую вершину ветви дерева при обходе всего узла сверху вниз и слева направо.
Query1*<?expYclEnd> refY,refSaveInd*;

refY — ссылка на узел потомки которого будут обходится. Может использоваться сокращенная ссылка.
RefSaveInd - ссылка на узел в который будут сохранен подиндекс потомка. Сохраняться будет только последний индекс потомка. Параметр обязательный. Данные узла доступны в системной переменной %queryData.

Команды обработки событий.

Команда **EventTrap (EvT)**.

Команда создает пользовательское событие.
EventTrap*<?expYcl> expName<(arg1<,argN>)> <,expName<(arg1<,argN>)>>*
EvT*<?expYcl> expName<(arg1<,argN>)> <,expName<(arg1<,argN>)>>*
expName — выражение задающее имя события. Имя события не может превышать 18 байт.
arg - аргументы которые будут переданы обработчикам этих событий.
По этой команде потоки ожидающие наступления этого события будут продолжены. С этого момента им будут доступны аргументы события.
По этой команде будут запущены в отдельных потоках все обработчики этого события и им будут переданы аргументы.
После завершения работы всех обработчиков события оно будет удалено.
Если нет ни одного обработчика этого события то событие будет ждать появления обработчика, затем будет обработано и удалено.

Команды обработки события.

Команда **EventCall (EvC)**.

Команда назначает обработчик завершения события.
EventCall*<?expYcl> expName=Prog <,expName=Prog >*
EvC*<?expYcl> expName=Prog <,expName=Prog >*
expName — выражение задающее имя события.
Prog — ссылка на программу обработчик события.

Если в момент выполнения этой команды событие существует то обработчик будет вызван и событие удалено.

Команда **EventWait (EvW)**.

Команда ожидать завершения события.
EventWait<*?expYcl*> *expName* <,*expName* >
EvW<*?expYcl*> *expName* <,*expName* >

Ожидает наступления события и когда оно происходит то возобновляется выполнение задания. С этого момента в программе становятся доступными аргументы переданные командой **EventTrap**. Если в момент выполнения команды **EventWait** событие существует то выполнение будет продолжено.
expName — выражение задающее имя события.

Команда **EventDelete(EvD)**.

Команда удаления события.
EventDelete<*?expYcl*> *expName*<,*expName*>
EvD<*?expYcl*> *expName*<,*expName*>
Удаляет события с заданными именами.

События могут быть вызваны процессами происходящими в системе. Это системные события они имеют префикс sys. События так же могут быть вызваны командой EventTrap в программе. Это пользовательские события и они имеют префикс usr.

Команды работы с файлами.

Ввод и вывод с файлом осуществляется через переменную B$.
Писать в файл команда :
Write<*?expYcl*> *pathFile*;
pathFile — выражение строковое значение которого является именем файла.
В файл выводятся переменные из массива B$. Если задан разделитель в системной переменной %dlmIO , то в поток ввода вывода между значениями выводится разделитель.

Читать из файла: команда
Read<*?expYcl*> *pathFile*;
pathFile — выражение строковое значение которого является именем файла.
Введенные значения будут помещены в массив B$. Если задан разделитель в системной переменной %dlmIO , то значения из потока выбираются через разделитель.

В имени файла переменной *pathFile* можно задать стандартные устройства ввода вывода.
- ST устройство STDOUT
- ER устройство STDERR
- SR устройство TCP сокет сервера, если обращение к языку выполнено из стандартного сервера данных.

9. События.

Еще одной эффективной парадигмой программирования являются события. Как правило языки программирования не содержат средств обработки событий. Они обычно вынесены в стандартные библиотеки. В MUMPS обработка событий в зачаточном состоянии и сводится к обработке ошибок. В MSH события включены в язык. События могут быть как системными, порожденными системой. Так и пользовательскими, порожденными пользователем с

помощью команды **EventTrap .** Аргументы этой команды передаются обработчикам этого события.

Системные события.

sysJobEnd — Завершение задания, в передаваемом аргументе номер завершенного задания. Это событие возникает в момент завершения любого задания.
Системные события работы с внешними устройствами.
Эти события инициируются командой EventTrap. Но обработчики будут вызваны только тогда когда событие будет завершено. При вызове обработчиков события в 1-ом аргументе возвращается идентификатор устройства и он будет положительным целым числом. Если произошла ошибка то возвращается в этом аргументе код ошибки, и он будет отрицательным целым числом. 2-й аргумент, если он есть, зависит от события.

sysRead — Чтение с устройства. Данные прочитанные с устройства передаются во 2-ом аргументе.
В качестве аргумента в команде EventTrap передается идентификатор файла.

sysOpenReadClose — Событие возникает при любом из этих событий.
В качестве аргумента в команде EventTrap передается имя файла.
Команда **EventCall** назначает событию программу обработчик. В момент возникновения события текущее задание будет прервано и будет выполнена программа обработчик в новом блоке Do. Как будто в этом месте встретилась команда Do с вызовом данного обработчика. Это значит что локальные переменные выполнявшейся программы в обработчике события будут недоступны.

Команда **EventWait** останавливает выполнение текущей программы и ожидает возникновение события. При возникновении события текущая программа продолжает работу и с этого момента в программе становятся доступными переданные аргументы. Новый блок Do не создается и поэтому все локальные переменные этого блока Do доступны после каоманды **EventWait.**

Если событие возникло до появления команд **EventCall** и **EventWait.** То событие будет обработано первой встретившейся командой **EventCall** или **EventWait**.

После обработки событие удаляется.

Команда **EventDelete** удаляет событие вместе с обработчиками.

10. Списки.

Любая переменная может быть списком. Список состоит из одного или нескольких значений. Значения могут быть любых типов в том числе и списками.
Создать список можно конструкцией:
ref={exp<,exp>}
ref — ссылка на переменную куда будет записан список.
exp — выражение которое станет соответствующим элементом списка.
Список может быть также возвращен командой **Query**.

Если ссылка *ref* является списком то в индексе она будет развернута в соответствующее количество индексов.
[*ref*]
Каждый элемент списка станет соответствующим индексом. В ссылке обычные индексы могут чередоваться со списками. Например переменная может выглядеть так :
[Af,[B,2],125,$1,5.6]

если переменные[B,2] и $1 являются списками то они будут развернуты в индексы в соответствии с размером списка.

11. Функции.

Функции могут быть встроенными и пользовательскими. Встроенные функции определены языком. Пользовательские разрабатываются программистом. Имя встроенной функции задается константой. В имени встроенной функции имеется префикс %.
Фактические параметры передаются в функцию по значению.

Встроенные функции.

Поиск метки в произвольном модуле.

%findLabel(*expLabel<,expMod>*)
expMod -выражение результатом которого является имя модуля.
expLabel -выражение результатом которого является имя искомой метки.
Функция возвращает либо 1- если такая метка существует, либо 0- если нет.
Например:
Set $1=**%findLabel**(*Lab,Mod*)**;**

Функции блокировки.

%sysLockR*(exp<,time>)*— блокировка по чтению с временем ожидания,
%sysLockW(*exp<,time>*)— блокировка по записи с временем ожидания,
%sysLockSt(*exp*)— статус блокировки

exp - выражение значение которого и будет Идентификатором блокировки
результат выражения не должен превышать 18 байт.
Если длинна больше то учитываться будут только первые 18 байт
time -выражение значение которого задает TimeOut в микросекундах (usleep), если оно не указано, то принимается = 1 .
Возвращаемое значение *ref*= результат блокировки
0- блокировка выполнена текущим заданием
<0-блокировка не выполнена
>0 —№ задания которое ранее установило блокировку. Блокировка не выполнена.

Функции обработки завершения задания.

%sysJobStatus(*exp*) – состояние задания 0-задание завершено, 1-задание выполняется.
%sysJobErr(*exp*) – код завершения задания.
%sysJobValue(*exp*) – возвращенное значение.
exp-выражение значением которого является идентификатор задания.

Файловые директории.

Файловые поддиректории содержащиеся в данной директории.

%sysReadDir(*expFile*)
expFile-выражение задающее имя директории
Результат выводится в массив %B$.

Файлы содержащиеся в данной директории.

%sysReadDirFile(*expFile*)

expFile-выражение задающее имя директории
Результат выводится в массив B$.

Рекурсивное содержание файловой директории.

Файловые поддиректории содержащиеся в данной директории.

%sysReadDirR(*expFile*)
expFile-выражение задающее имя директории
Результат выводится в массив B$.

Файлы содержащиеся в данной директории.

%sysReadDirFileR(*expFile*)
expFile-выражение задающее имя директории
Результат выводится в массив B$.

Создать файловую директорию.

%sysCreateDir(*expFile*)
expFile-выражение задающее имя директории

Удалить файловую директорию.

%sysDeleteDir(*expFile*)
expFile-выражение задающее имя директории

Удалить файл.

%sysDeleteFile(*expFile*)
expFile-выражение задающее имя файла

Функции возвращают количество обработанных файлов.

Компиляция файла.

Функция компиляции файла:
%sysCompile(*nameMod*);
nameMod — выражение строковое значение которого является именем модуля msh. Модуль ищется в стандартном каталоге для исходных программ ./msh
В результате трансляции будет создан файл команд виртуальной MSH машины в каталоге mvm.
Код возврата является 32 битным вектором:
компонента 0- код возврата, либо 0, либо отрицательное число(код ошибки),
компонента 1- номер строки где была завершена обработка,
компонента 2- номер позиции где была завершена обработка.
Если возврата =0, то номер строки это последняя обработанная строка и позиция.
Например транслируем модуль byxUsrUI:
Set $1=**%sysCompile**(byxUsrUI),$2=$1.**%v32**(0),$3=$1.**%v32**(1),$4=$1.**%v32**(2);
$2- содержит код возврата, $3 – номер строки где была завершена обработка, $4 – номер позиции где была завершена обработка.

2. Пользовательские функции.

Пользовательские функции представляют из себя обычные подпрограммы и имеют формат аргументов команды Do но с обязательными скобками.
expMod.expLabel(<expArg <,expArg>>)
.expLabel(<expArg <,expArg>>)
expLabel(<expArg <,expArg>>)

expMod — выражение задающее имя модуля.
ExpLabel - выражение задающее точку входа (имя подпрограммы)
expArg — выражение задающее аргумент.

12. Предопределенные свойства.

Системные свойства.

Системные свойства дают дополнительную информацию о системе.
%err — код завершения последней операции.
%this — Индекс Объекта.
%queryKey — ссылка последнего обращения к данным в итераторе.
%queryData — значение последнего обращения к данным в итераторе.
%Data — значение последнего обращения к данным в команде Get.
%eof – при завершении итератора становится равным 1, иначе 0,%statData — состояние переменной к которой было последнее обращение.
%dmtIO — разделитель полей в операциях ввода вывода, устанавливается для каждого задания.
%nameMod — имя исполняемого модуля.
%idDiv — текущее устройство ввода-вывода.
%isTr- статус транзакции. Выполняется ли код внутри транзакции.
%idJob — ID выполняющегося задания.
%idDo — ID блока Do.
%statDo — Статус выполнения программы.

Свойства любых переменных.

У каждой переменной есть предопределенные свойства
*refData***.%type** - тип переменной
*refData***.%byte** - размер переменной в байтах в строковой форме
*refData***.%size** - размер переменной в символах в строковой форме
*refData***.%stat** - состояние переменной.
Тип результата устанавливается в соответствии с типом свойства.

Методы Данных.

Конструктор объекта.

refData**.%objNew*(Type)***

refData-ссылка на данные дерева.

Type -тип переменной.

Например:

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

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

Обход данных.

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

expLabel – метка функции обработки вершины,

expMod – модуль функции обработки вершины.

IsProp -будет ли метод обходить свойства узла, если ~=0 , то да, при обходе узла сначала будут обходиться свойства узла , а только затем его потомки.

Вызов программы обработки будет осуществляться в новом контексте и в задании текущей программы.

Этой функции в качестве аргументов передаются :

1. признак свойства, если этот аргумент =1, то передано свойство узла иначе передан узел,
2. последний индекс узла,
3. данные в узле,
4..N полный путь к узлу.

Программа обработки должна возвращать 0, для продолжения.

Чтение Ini файлов.

Выполняется как метод Индексированых данных. *pathFile* — выражение строковое значение которого является именем файла. Чтение происходит в индексированую переменную refData.

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

Например:

Do [1].**%iniRead(«srv.ini»);**

Запись Ini файлов.

Выполняется как метод Индексированых данных. *pathFile* — выражение строковое значение которого является именем файла. Запись происходит в файл *pathFile*.

refData**.%iniWrite(***pathFile***)**

Например:

Do [1]**.%iniWrite(«srv.ini»);**

Строковые свойства.

Перед выполнением этих свойств переменная преобразуется к строковому формату.

Действия над символами строки.

StartInd- индекс символа с которого надо выполнять указанное действие.

Count — количество символов участвующих в действии.

1. Получить *Count* символов строки начиная с символа с индексом *startInd*. Если *Count* не задан то считается что он равен 1:
   refData.**%strGet**(*startInd <,Count>*)
   например: [j1,j2]=[i1,i2,i3].**%strGet**(5,3)
2. Получить код символа строки с индексом *startInd*
   refData.**%strCode**(*startInd*)
   например: [j1,j2]=[i1,i2,i3].**%strCode** (5)
3. Снять с вершины строки Count символов
   refData.**%strPop**(*Count*)
   например: [j1,j2]=[i1,i2,i3].**%strPop**(2)
4. Вставить в строку символы начиная с символа с индексом *startInd*

refData.**%strIns**(*startInd*)
например: [i1,i2,i3].**%strIns**(7)=[j1,j2]
или $2=[i1,i2,i3].**%strIns**(7,[j1,j2])

5. Заменить в строке *Count* символов:
refData.**%strSet**(*startInd <,Count>*)
например: [i1,i2,i3].**%strSet**(4,3)=[j1,j2]

6. Удалить из массива *Count* символов начиная с символа с индексом *startInd*
refData.**%strDel**(*startInd <,Count>*)
например:
[j1,j2]=[i1,i2,i3].**%strDel**(5,2)

7. Найти подстроку *subStr* начиная с символа с индексом *startInd* если *startInd* не задан то поиск начинать с начала строки :
.**%strFind**(*subStr<,startInd>*)
Результат: позиция первого искомого символа.
например:
[j1,j2]=[i1,i2,i3].**%strFind**(ABC,2)

8. В строке символы *charSource* заменить символами *charRepl* только 1 вхождение если *charRepl* не задан то символы *charSource* удалить из строки:
refData.**%strRepl**(*charSource<,charRepl>*)
например:
[j1,j2]=[i1,i2,i3].**%strRepl**(ABC,123)

9. В строке все символы *charSource* заменить символами *charRepl* все вхождения если *charRepl* не задан то все символы *charSource* удалить из строки:
refData.**%strReplALL**(*charSource<,charRepl>*)
например:
[j1,j2]=[i1,i2,i3].**%strReplALL** (ABC,123)

10. Заменить символы *charSource* в строке *charStr* на символы *charRepl.*
refData.**%strUpDate**(*charStr,charSource<,charRepl>*)
например:
[j1,j2]=[i1,i2,i3].**%strUpDate**(ABCDгщы,BCг,bcГ)
Результат : AbcDГщы

Действия над полями строки.

Если *Count* не задан то считать его равным 1. *delimiter* разделитель полей в строке.

1. Получить *Count* полей строки начиная с поля *startInd* , если *Count* не задан то считать *Count*=1:
refData.**%fieldGet**(*delimiter,startInd <,Count>*)
например:
[j1,j2]=[i1,i2,i3].**%fieldGet**(',',5,3)

2. Добавить поля в строку:
refData.**%fieldPush**(*delimiter*)
например
[i1,i2,i3].**%fieldPush**('.')=[j1,j2]
Если добавляемая строка не указана, то добавляется только разделитель.
Например:
[i1,i2,i3].**%fieldPush**('.')

3. Снять с вершины строки *Count* полей если *Count* не задан то считать *Count*=1.
refData.**%fieldPop**(*delimiter<,Count>*)
например
[j1,j2]=[i1,i2,i3].**%fieldPop**('.',2)

4. Вставить в строку поля
refData.**%fieldIns**(*delimiter,startInd*)
например:
[i1,i2,i3].**%fieldIns**('/',7)=[j0,j1,j2]
5. Заменить в строке *Count* полей начиная с поля *startInd*
refData.**%fieldSet**(*delimiter,startInd ,Count*)
например
[i1,i2,i3].**%fieldSet**('-',4,3)=[j1,j2]
$3.**%fieldSet**('-',4,3)='AB'

6**.** Удалить в строке *Count* полей начиная с поля *startInd*
refData.**%fieldDel**(*delimiter,startInd <,Count>*)
например
[j1,j2]=[i1,i2,i3].**%fieldDel**('-',5,2)

7. Определить количество полей в строке.
refData.**%fieldCount**(*delimiter*)
например
[j1,j2]=[i1,i2,i3].**%fieldCount**('-')

Действия над списками.

1. Получить *Count* элементов списка начиная с *startInd*
refData.**%listGet**(*startInd <,Count>*)
например [j1,j2]=[i1,i2,i3].**%listGet**(5,3)
2. Добавить к списку другой список
refData.**%listPush**
например
[i1,i2,i3].**%listPush**=[j1,j2]
3. Снять с вершины списка *Count* элементов списка
refData.**%listPop**(*Count*)
refData.**%listPop -** Снять с вершины списка 1 элемент
например
[j1,j2]=[i1,i2,i3].**%listPop**(2)
4. Вставить в список на место элемента *startInd* другой список
refData.**%listIns**(*startInd*)
например
[i1,i2,i3].**%listIns**(7)=[j1,j2]
5. Заменить в списке *Count* элементов начиная с элемента *startInd*
refData.**%listSet**(*startInd <,Count>*)
например
[i1,i2,i3].**%listSet**(4,3)=[j1,j2]
6. Удалить из списка *Count* элементов начиная с элемента *startInd*
refData.**%listDel**(*startInd <,Count>*)
например
[j1,j2]=[i1,i2,i3].**%listDel**(5,2)

7. Определить количество элементов в списке.
refData.**%listCount**
например
[j1,j2]=[i1,i2,i3].**%listCount**

Действия над массивом данных.

1. Добавить в массив дополнительный элемент.
refArr.**%arrPush**()
например
Set m$.**%arrPush**=ABC
Set m$.**%arrPush(**ABC,RT,125)
Set $1=m$.**%arrPush(**ABC,RT)
refArr – сокращенная ссылка на массив.
ABC,RT,125 – добавляемые элементы.
2. Снять с вершины массива элемент
refArr.**%arrPop**
например
[j1,j2]=m$.**%arrPop**
или удалить с вершины элемент
m$.**%arrPop**

13. Векторы.

В данных могут хранится целочисленные вектора. Количество компонент вектора зависит от размерности компоненты.

Размерность компоненты вектора может быть 64 бита, 32 бита, 16 бит и 8 бит. Соответственно размерность этих векторов будет 2, 5, 11 и 22. Вектора могут быть значениями как со знаком, так и без знака. Отсчет координаты вектора ведется от 0.

64 битные вектора.

Обращение к знаковой компоненте вектора:

refArr.**%v64**(n);

n — номер координаты вектора.

Обращение к беззнаковой компоненте вектора:

refArr.**%vu64**(n);

n — номер координаты вектора.

Размерность вектора 2.

32 битные вектора.

Обращение к знаковой компоненте вектора:

refArr.**%v32**(n);

n — номер координаты вектора.

Обращение к беззнаковой компоненте вектора:

refArr.**%vu32**(n);

n — номер координаты вектора.

Размерность вектора 5.

16 битные вектора.

Обращение к знаковой компоненте вектора:

refArr.**%v16**(n);

n — номер координаты вектора.

Обращение к беззнаковой компоненте вектора:

refArr.**%vu16**(n);

n — номер координаты вектора.

Размерность вектора 11.

8 битные вектора.

Обращение к знаковой компоненте вектора:

refArr.**%v8**(n);

n — номер координаты вектора.

Обращение к беззнаковой компоненте вектора:

refArr.**%vu8**(n);

n — номер координаты вектора.

Размерность вектора 22.

14. Функции библиотеки Win.

Библиотека **Win** позволяет создавать и манипулировать визуальными компонентами окон. Данная библиотека построена на основе GTK+3.0 и тесно с ней связана. Свойства объектов библиотеки **Win** в основном соответствуют свойствам Widget библиотеки GTK.

Библиотека **Win** перед использованием должна быть инициализирована функцией:

%winInit()

после использования ресурсы библиотеки должны быть освобождены функцией:

%winFree()

Остальные функции вызываются между ними.

Функция:

%winNew(*type,nameObj<,namePar*ent*,Arg...>***)**

Создает объект и помещает его в иерархию визуальных объектов,

type — тип создаваемого объекта,

nameObj — имя нового объекта,

nameParent — имя предка, может быть пустым, если предка нет,

Arg — параметры необходимые для создания объекта. Список параметров зависит от создаваемого объекта.

Функция:

%winSetAtr(*typeAtr,nameObj,dataSet<,Arg...>***)**

Устанавливает свойство объекта в значение dataSet.

typeAtr — тип атрибута который надо изменить,

nameObj — имя объекта,

dataSet — новое значение атрибута,

Arg — дополнительные параметры необходимые при изменении атрибута. Список параметров зависит от создаваемого объекта.

Функция:

%winGetAtr(*typeAtr,nameObj<,Arg...>***)**

Чтение атрибута объекта.

typeAtr — тип атрибута который надо получить,

nameObj — имя объекта,

Arg — дополнительные параметры необходимые для доступа к атрибуту. Список параметров зависит от создаваемого объекта.

Функция:

%winIniToAtr(*typeAtr,nameIniFile<,nameIniFile...>***)**

Читает Ini файл и для объектов перечисленных в этом файле для атрибута устанавливает соответствующее значение указанное в этом файле.

nameIniFile — имя Ini файла,

typeAtr — атрибут значение которого необходимо установить.

Функция:

%winAtrToIni(*typeAtr,nameIniFile***)**

Записывает в Ini файл для объектов перечисленных в этом файле и указанного атрибута значение соответствующего атрибута объекта.

nameIniFile — имя Ini файла,

typeAtr — атрибут значение которого необходимо установить.

Если необходима связь с сервером, то надо открыть канал связи.

Функция:

%winOpenChannel(*iniPipe***)**

iniPipe — файл содержащий настройки канала.

Для отображения win компонентов необходимо запустить главный цикл приложения функцией:

%winStart(*nameObj***)**

nameObj — главное окно приложения.

15. Структура модуля.

Модуль состоит из строк. Строка может начинаться с метки. Тогда после метки стоит двоеточие.Далее через пробел могут следовать команды. Команды отделяются друг от друга точкой с запятой ';' .

16. Объекты.

В MUMPS объектов нет. Но ООП получило повсеместное распространение. Основная масса современных языков программирования так или иначе поддерживает ООП. MSH не исключение. Для того чтобы было ООП язык должен иметь для начала объекты. В общем случае описание объект состоит из декларативной части и части реализации. MSH не поддерживает декларации переменных, значит декларативной части описания объектов не будет. Существует описание объекта в MSH в виде части реализации. В MUMPS код программы представлен в виде программных модулей. Точки входа в модуле являются подпрограммами и функциями. Программный модуль в MSH и взят в качестве описания объекта (класса). Метки модуля являются точками входа этого модуля. Точки входа в этом случае являются публичными свойствами get этого класса. Публичными свойствами set этого класса будут та же точка входа как и get модифицированная определенным символом. В качестве такового в MSH выбран символ точка «.». У свойства в MSH кроме доступа get и set есть еще операция Kill -удаление свойства. В этом случае имя свойств модифицируется префиксом «..». Наследование классов в MSH является множественным и реализовано с помощью команды **Parent**. Классы перечисленные в этой команде в порядке следования являются предками данного класса. Во время выполнения программы при обращении к свойству объекта его описание сначала ищется в исходном модуле затем в модулях перечисленных в команде **Parent в порядке их следования. Более раннее положение** модуля в команде дает ему больший приоритет. Защищенные свойства реализуются с помощью системного свойства %this. Объекты в MSH могут находиться только в дереве. В массиве могут храниться только примитивные типы данных.

Методы класса так же являются точками входа в модуле класса. Создаются объекты с помощью стандартного свойства **%objNew**.

Например имеем класс Person со свойством Age. В модуле класса есть точки входа «Age» и «.Age».

В переменной [1,2] создадим объект.

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

Обращение к свойству Age объекта Person по записи.

Set [1,2].Age=50;

Обращение к свойству Age объекта Person по чтению.

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

Эти команды транслятор преобразует в обращения к подрограммам. Но это не единственный способ обращения к свойствам. По записи к свойсту Age можно обратиться и через метод.

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

В MSH модуль выполняет двоякую роль. Кроме модуля класса он может трактоваться и как модуль программ. В этом случае к точкам входа обращаются как к подпрограммам. Предположим в модуле Person есть метка ABC. Тогда к ней можно обратиться как к программе. Правда в этом случае переменная %this будет пустой.

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

Так можно организовать классовые методы.

Модули могут выступать в качестве Типа(Класса) объекта. Имя модуля соответствует типу объекта и не должно превышать 18 байтов. Имя свойства по чтению должно быть меткой в модуле. Свойство по записи должно быть меткой совпадающей с именем свойства с префиксом <.> .Удаление свойства выполняется в модуле с меткой имеющей префикс <..>

Наследование обеспечивается командой **Parent**. Любая переменная может стать объектом для этого достаточно присвоить ее предопределенному свойству .**%objNew имя типа** (модуля). Например имеется тип (модуль) Org и свойство этого типа Arg.
Создаем объект этого типа:
Set [An,12].**%objNew=Org; Тогда обращение к свойству Arg по записи и по чтению будет** выглядеть так:
[An,12].Arg=[An,12].Arg+1;

17. Замечания.

Команды не имеющие аргументов могут не иметь признака конца команды «;». Это команды : **If, Else, End, Case, Break.**

18. Заключение.

Считаю что описанный язык будет отвечать поставленным целям. Приведенное множество команд позволяет реализовать все современные концепции программирования. Наличие команд создания заданий и обработки событий позволяет упростить процесс программирования. Управление данными в языке позволяет исключить большое количество ошибок при разработке программ.

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