MUMPS Compiler

Final Report

ITN162 Sem 2 2008

Ray Newman

n6942962

13 November 2008

Supervisor Diane Corney

Coordinator Ernest Foo

Table of Contents

On the CDROM

1). ISO/IEC 11756:1999 - ISO Standard MUMPS
2). Sources One Software MUMPS implementation
3). Extreme Database Programming with MUMPS
4). MUMPS Compiler Routines (save format)
5). MUMPS Compiler Listing for routine FBENCH
6). MUMPS Compiler Final Report (this document)

INTRODUCTION

According to Wikipedia, MUMPS (Massachusetts General Hospital Utility Multi-Programming System) was created during 1966 and 1967 in Dr Octo Barnett's animal lab at Massachusetts General Hospital; like Unix a few years later, built on a spare DEC PDP-7.

MUMPS was originally for use in the healthcare industry. It was designed to make writing database-driven applications easy while simultaneously making efficient use of computing resources. MUMPS was adopted as the language-of-choice for many healthcare and financial information systems databases (especially those developed in the 1970s and early 1980s) and continues to be used by many of the same clients today.

It is currently used by the world's largest electronic health record systems as well as by multiple banking networks and online trading investment services.

By the early 1970s, there were many and varied implementations of MUMPS on a range of hardware platforms. The most widespread was DEC's MUMPS-11 on the PDP-11, and Meditech's MIIS. In 1972, many MUMPS users attended a conference to standardise the now fractured language. This also created the MUMPS Users Group (MUG) and MUMPS Development Committee (MDC). These efforts proved successful; a standard was complete by 1974, and was approved, on September 15, 1977, as ANSI standard, X11.1-1977. ***MUMPS was the third ANSI standard language after COBOL and FORTRAN***.

THE LANGUAGE

The age and origin of the language contributes to its format. While COBOL and FORTRAN of the era relied on punched cards for input and therefore had a structure that looked like a deck of cards, MUMPS due to its DEC base was a terminal or line based language.

Other aspects of the language derived from its origin include its compact and abbreviated nature and its lack of internal standards; for example command argument formats vary widely between commands and the language contains an "indirection" ability that allows arguments to be supplied indirectly where a variable may contain the names of the variables to be acted on.

As the language was initially designed to be interpretative in a very limited machine (limited to 56 KBytes of memory and disks that were less than one MByte in size, all commands could be abbreviated to their initial letter (even in the case where two different commands have the same initial letter). To save on extra scoping commands, the *For, If* and *Else* commands were limited to the current line of the program.

The number of spaces between commands and arguments is significant; again to make the most of available resources by using the spaces as delimiters.

Why MUMPS

I first used MUMPS in 1976 while employed by Digital Equipment Corporation and moved exclusively to using it in 1980 when I established a computer bureau in Northern NSW. As an indication of the efficiency of the language, on that machine we could run up to 20 on-line users. The machine was a DEC PDP 11/23 (a 10 KHZ processor with 128 KBytes RAM and 20 MBytes of disk). Some of the machines I've installed over the last few years were one million times faster with 16 thousand times the memory and many thousand times the disk capacity.

MUMPS is about database. The database is a persistent sparse multi-dimensional array where both indexes and data are variable length strings. A simple example is:

```
^PC("ALBERTON","Q")=4207
^PC("ALBERTON","S")=5014
^PC("ALBERTON","T")=7263
^PC("ALBERTON","V")=3971
^PC("ALBION","Q")=4010
^PC("ALBION","W")=6055
```

This represents a portion of the Australian post code table. This is a two level table where the first level is the location and the second level (which also contains the data) is the state. To create another entry for Albion Vic, we would enter the command:

```
SET  ^PC("ALBION","V")=3020
```

This would then appear in the list after Albion, Qld. To remove the entry for Alberton SA, use the command:

```
KILL ^PC("ALBERTON","S")
```

The following code fragment lists locations commencing with ALBERT with states and postcode.

```
Set (Tst,Loc)="ALBERT"
For  Set Loc=$order(^PC(Loc)) Quit:$extract(Loc,1,$length(Tst))'=Tst  Do
. Write Loc Set St=""
. For  Set St=$order(^PC(Loc,St)) Quit:St=""  Write ?20,St,?35,^(St),!
```

The output would look like:

```
ALBERTON    Q    4207
            S    5014
            T    7263
            V    3971
```

The MUMPS database is a heavy weight performer in a light weight design. As the database is schema-less, there is no necessity to reload for a design change as there is with the traditional relational model.

For more discussion on using the MUMPS database, see the paper titled Extreme Database Programming on the accompanying CD and *url*

http://www.slideshare.net/george.james/mumps-the-internet-scale-database-presentation

MUMPS is about simplicity. It has only one data type; that is string. Conversion to a numeric or truth value is automatically performed as required. For example:

```
Write “4 Oranges”+“3 Apples”      gives 7
If “4 Oranges”=”4 Apples”         is false - but
If +”4Oranges”=+”4 Apples”        is true - or
If “4 Oranges”                    is true - but
If “Four Oranges”                 is false
```

The single data type makes producing html and javascript quite easy and hence the language is quite useful in writing web servers as no data conversion is required.

The language also has built in formatting with functions such as $Justify(value,field_size,decimal_places) and $FNumber(). It also includes extensive string handling functionality.

THE PROJECT

The purpose of this project is to start to re-implement the MUMPS language compiler properly doing full analysis of the syntax and semantics of the language in the process.

The scope of the project includes full analysis of the language as described in ISO/IEC 11756:1999 and as implemented by One Software (my development organisation) in 1999 and a start on coding.

It is not expected that the compiler will be complete as part of this project.

As the language contains constructs not seen in any other language, it is expected that the analysis using modern day tools will be quite time consuming.

Due to an escalating price for the MUMPS environment, in 1999 we (at One Software) decided to write our own implementation. This was written "under the whip" - we had committed to install on the new platform within twelve months.

As a result of the time constraints, the current implementation is not very well written and given the opportunity this year, I have had a look at re-writing the compiler portion of the implementation using modern tools and techniques.

Although the ANSI standard for MUMPS has been allowed to lapse, the most recent standard (ISO/IEC 11756:1999, re-affirmed on 6 January 2005) still defines essentially the same language. This was chosen as the standard to apply to this project. Although the original implementation was done to the ANSI/MDC X11.1-1995 standard, a comparison showed very little difference in the definitions.

PROBLEMS

The first problem encountered was that the location and number of spaces in the source code is **sometimes** significant. That is a line commences with either a label or a space; a label (and its parameter list if any) must be followed by one or more spaces. Each command must be separated from its arguments by exactly one space and if the command has no arguments, it must be separated from the next command by at least two spaces unless the original command is the last thing on the line when no spaces are required. Also, no argument or argument list may contain an unquoted space.

Another problem is the ability to add a post condition to most commands (except for *If, Else* and *For*) where the syntax is *command colon~ condition*. The condition must be true for the command to be executed. In the case of the *Goto, Do* and *Xecute* commands, the arguments may also have a post condition. In addition, conditions and any argument (except arguments to the *For* command) may be indirect. The syntax is @ expression where the expression must be evaluated and then compiled at run time.

As mentioned previously, all commands may be abbreviated to one character. This gives rise to a problem with the *Hang* (equivalent a sleep) and *Halt* commands. Both may be abbreviated to simply *H*. The parser then must work out which command it is by the presence or absence of arguments to the command.

The double use of the equal sign for assignment and comparison also gives rise to confusion. The form: `Set A=B=C` means compare `B` with `C` and assign the result (1 or 0) to `A`. While the form: `Write A=B=C` means compare `A` with `B` and compare the result (1 or 0) with `C` then write out the result (1 or 0).

Another complication is the inclusion of the *Xecute* command which takes an expression (which may be indirect) and compiles and executes it.

At run-time there is the concept of the "naked indicator". This is a pointer to the last location accessed from the database and may be used as a short-hand method of accessing the database. The contents of this indicator is very important and must be predictable. This means the construct If true or expression where expression contains directly or indirectly a database access cannot be compressed at compile time; or in code:

```
If 1!^DB("test",6) Write ^(4) ; Must write out ^DB("test",4)
```

Note that the exclamation mark (!) above represents *OR* in MUMPS.

On the other hand, MUMPS has some very simple rules; like all evaluations are strictly left to right (with no precedence) and accessing the database is exactly the same as accessing a local variable such that the construct *NAME* is a local variable and *^NAME* is a global variable (database).

The concept of a "local" variable is what other languages (such as **c**) would call a global; in that the variable is (usually) available to all

programs run in the current user space; this unfortunately removes the possibility of any reasonable compile time optimisation of variable storage. The variables are just “there” when a program is loaded and when it is exited.

IMPLEMENTATION - SCANNER

Because of some of the inconsistencies in the definition of MUMPS, we are unable to use some of the more modern compiler tools such as YACC.
We are, however, able to use modern design techniques.

We commenced using EBNF (Extended Backus–Naur Form) to describe MUMPS. The result was a separate definition for the argument to each command. Then this was compressed to have one definition to cover all command arguments which resulted in basically an "anything".

We settled for a scanner that output only the terminal symbols *intlit, name, number, stringlit* and each binary operator (and other punctuation). This gave a simple scan code stream.

MUMPS Syntax in EBNF is given on the next two pages with the scan codes used on the following page.
The following is partial scanner output:

```
   5  write !,"Scanning routine ",RouNam,!
SPACE NAME=write SPACE SCOR SCCOMA STRLIT="Scanning routine " SCCOMA NAME=RouNam SCCOMA
SCOR EOL

   6 ;Say what we are doing
EOL

   7 LOOP for  do  do GETLINE quit:'$Data(LineBuf)
NAME=LOOP SPACE NAME=for SPACE SPACE NAME=do SPACE SPACE NAME=do SPACE NAME=GETLINE SPACE
NAME=quit SCCOLON SCNOT SCDOLR NAME=Data SCLP NAME=LineBuf SCRP EOL

   8 ;Loop for all lines
EOL

   9  . write !!,$Justify(LineNo,4)," ",LineBuf,!
SPACE LEVEL SPACE NAME=write SPACE SCOR SCOR SCCOMA SCDOLR NAME=Justify SCLP NAME=LineNo
SCCOMA INTLIT=4 SCRP SCCOMA STRLIT=" " SCCOMA NAME=LineBuf SCCOMA SCOR EOL

  10 ;. Display the line being scanned
EOL
```

MUMPS Syntax

Definitions

controls are the ASCII codes 0 to 31 and 127.
digits are any numeric character '0' ..9.
graphics are any ASCII code 0 to 127 except controls.
letters are any alpha character 'A' .. 'Z', 'a' .. 'z'.
space is the ASCII 32 space character.

Terminal Symbols

binaryop	=	_ | + | - | * | / | # | \ | ** | = | < | > |] | [|]] | & | ! | ? | '= | '< | '> | '] | '[| ']] | '& | '! | '?
intlit	=	digit digit*
name	=	'%' | letter (digit | letter)*
number	=	digit (digit)* ['.' digit (digit)*] | '.' digit (digit)*
strlit	=	In words, a string literal is bounded by quotes and contains any string of printable characters, except that when quotes occur inside the string literal, they occur in adjacent pairs. Each such adjacent quote pair denotes a single quote in the value denoted by strlit, whereas any other printable character between the bounding quotes denotes itself. An empty string is denoted by exactly two quotes.
unaryop	=	+ | - | '

EBNF Grammar

Routine	→	Line { Line }
Line	→	[Label] Spaces Dots [Command { Spaces Command }]
Label	→	name | intlit ['(' NameList ')']
Dots	→	{ '.' [Spaces] }
Command	→	name PostConditional space [Args] | ('F[OR]' Var '=' Expr [':' Expr [':' Expr]] { ',' Expr [':' Expr [':' Expr]] })
PostConditional	→	[':' Expr]

```
Spaces      →  space { space }

Args        →  Arg { “,” Arg }
Arg         →  [ '@' ] Expr PostConditional
               | EntryRef [ Actuallist ] Postconditional
               | EntryRef [ Actuallist ] [ '::' Expr ]
               | name
               | NameList { ':' 'S[ERIAL]' | 'T[RANSACTIONID]'}
               | '(' NameList ')'
               | Var '=' Var
               | Format
               | [ '*' ] Expr
               | [ '*' ] Var [ '#' Expr ] [ ':' Expr ]
               | '(' VarList ')'
               | [ '+' | '-' ] Var
               | SetDest | ( '(' SetDest { ',' SetDest } ')' ) '=' Expr
               | Expr ':(' Expr ':' Expr ')' [ ':' Expr [ ':' Expr ] ]
               | Expr [ ':' [ Expr | ( '(' Expr { ':' Expr } ')' ) ] ] [ ':' Expr ]
               | Expr ':' Expr

Actuallist  →  '(' [ Actual ] { ',' Actual } ')'
Actual      →  ('.' name ) | Expr
EntryRef    →  Label [ '+' Expr ] [ '^' [ '|' name '|' ]  name ] |
                  '^' [ '|' name '|' ] name
Format      →  strlit | '!' | '#' | '?' Expr | '/' name [ '(' Expr { ',' Expr } ')' ]
Function    →  '$' ( '$' EntryRef | '&' name | name ) [ '(' ExprList ')' ]
               | ( '$' name '(' Expr ':' Expr { ',' Expr ':' Expr } ')' )
Expr        →  { unaryop } ( number | strlit | Var | '(' Expr ')' | Function )
               { binaryop Expr }
ExprList    →  Expr {‘,’ Expr }
NameList    →  name { ‘,’ name }
SetDest     →  Var | '$' name [ '(' Var ',' [ ExprList ] ')' ]
VarList     →  Var { ‘,’ Var }
Var         →  [ ‘^’ [ ‘|’ name ‘|’ ] ] name [ ‘(‘ ExprList ‘)’ ]
```

SCAN CODES

```
-1=ERROR                Generic error
0=EOL                   End of Line (input)
                --- Start of Operators ---
1=SCCAT                 Concatenate (underscore)
2=SCPLUS                Plus
3=SCMINUS               Minus
4=SCMUL                 Multiply (asterisk)
5=SCDIV                 Divide (slash)
6=SCMOD                 Modulus (hash)
7=SCINT                 Integer Divide (back slash)
8=SCPWR                 Power (asterisk asterisk)
9=SCEQ                  Equal
10=SCLT                 Less than (left caret)
11=SCGT                 Greater than (right caret)
12=SCFOL                Follows (right square bracket)
13=SCCON                Contains (left square bracket)
14=SCSA                 Sorts after (right square bracket right square bracket)
15=SCAND                And
16=SCOR                 Or (exclamation mark)
17=SCPAT                Pattern Match (question mark)
18=SCNOT                Not (single quote)
19=SCNEQ                Not Equal (single quote equal)
20=SCNLT                Not Less Than (single quote left caret)
21=SCNGT                Not Greater Than (single quote right caret)
22=SCNFOL               Not Follows (single quote right square bracket)
23=SCNCON               Not Contains (single quote left square bracket)
24=SCNSA                Not Sorts After (single quote right square bracket x 2)
25=SCNAND               Not And (single quote and)
26=SCNOR                Not Or (single quote exclamation mark)
27=SCNPAT               Not Pattern Matches (single quote question mark)
                --- End of operators ---
28=SCLP                 Left Parenthesis
29=SCRP                 Right Parenthesis
30=SCCOLON              Colon
31=SCCOMA               Coma
32=SCDOLR               Dollar Sign
33=SCCARET              Caret (up arrow)
34=SCINDIR              Indirection (commercial at sign)
35=SCVB                 Vertical Bar
36=SPACE                Space
37=LEVEL                Dot or Period
38=INTLIT               Integer Literal (followed with = and value)
39=NAME                 Name or identifier (followed with = and value)
40=NUMBER               Numeric literal (followed with = and value)
41=STRLIT               String literal (followed with = and value)
42=EXPR                 Top node of an expression
43=MUMPSVAR             Top node of a MUMPS variable
44=ENTRYREF             Top node of an entry reference
45=FUNCTION             Top node of a function
```

IMPLEMENTATION - PARSER

The next step was to parse this stream. After much discussion and experimentation, the first pass of the parser was written to examine each line (MUMPS is a line oriented language) to determine the label (if any), the formal parameters (if any) and the execution level of that line. The remainder of the line is then parsed to the extent of recognising each command but leaving the arguments and post conditions in scan form.

The commands may be separated one from the next by using the fact that each command (with it arguments) contains exactly one un-quoted space which is used to separate the command word from its arguments (if any).

This intermediate structure is stored in a MUMPS database where subsequent passes simply add elements as they are parsed. The final pass is to take the fully parsed information from the database and output compiled code.

Arguments to MUMPS commands are broadly formed from the following:

1). Expression including literals, functions, variables and sub-expressions.
2). Entry references including MUMPS and 'external' routines.
3). Names of Variables being local (to this process) or global (on-disk)

Commands that require Expressions as all or part of the argument are: *Break, Close, For, Hang, If, Open, Quit, Set, Use, View, Write* and *Xecute*.

In addition post conditions are expressions and may apply to the commands *Break, Close, Do, Goto, Halt, Hang, Job, Kill, Lock, Merge*, *New, Open, Quit, Read, Set, Use, Write* and *Xecute* and to individual arguments to the commands *Do, Goto* and *Xecute*.

Commands that require Entry references as all or part of the argument are: *Do, Goto* and *Job*. In addition an Entry reference may be part of a function definition which forms part of an expression.

Commands that require Names of Variables as all or part of the argument are: *For, Kill, Lock, Merge, New, Read* and *Set*. In addition variables may form part of expressions.

Within the arguments of a single command, the types required vary also. For example the *Kill* command has two forms as follows:

1). Inclusive Kill; arguments are any local or global variables that are to be un-defined (or killed).

2). Exclusive Kill; arguments must be un-subscripted local variables - these being the only local variables that are NOT to be un-defined (or killed).

Within the parser, the commands were grouped as follows:

Break, Quit - zero or one argument (expression)

Close, Hang - one or more arguments (expression)

Do - zero or more arguments (entry references)

Else, Halt - no argument permitted

For (no indirection or postconditional permitted) - no argument or a variable followed by equals and one or more arguments of the form: expression (initial value) optionally followed by colon expression (increment) optionally followed by colon expression (terminating value).

Goto - one or more arguments (entry references) without actual lists

If (no postconditional permitted) - zero or more arguments (expression)

Job - one or more arguments (entry references) without postconditionals

Kill - zero or more arguments (variables) or one or more unsubscripted local variables enclosed in parenthesis separated by commas

Lock - zero or more arguments (variables) optionally preceded by either

plus (+) or minus (-) or one or more variables enclosed in parenthesis separated by commas

Merge - one or more arguments consisting of destination variable equals source variable

New - zero or more arguments (unsubscripted local variables) or one or more unsubscripted local variables enclosed in parenthesis separated by commas

Open - one or more arguments consisting of three to five sub-arguments (expressions) of the form arg1 ':(' Sarg2 ':' Sarg3 ')' [':' Sarg4 [':' Sarg5]]

Read - one or more arguments each of the form ['*'] variable ['#' expression] [':' expression] or formatting command which is of the form *number, ?number, # or !

Set - one or more arguments each of the form: one or multiple enclosed in parenthesis of a variable, special variable $Ecode, $Estack, $Etrap, $Key, $X or $Y or the function $Extract() or $Piece() followed by equals and a source expression

Use - one or more arguments consisting of one or more sub-arguments (expressions) of the form Sarg1 ['('] [':' Sarg2] [':' Sarg3] ... [')'] [':' last Sarg] where parenthesis are permitted for two sub-arguments and required for four or more

View - one argument consisting of two expressions separated by colons

Write - one or more arguments each of the form ['*'] expression ['#' expression] [':' expression] or formatting command which is of the form *number, ?number, # or !

Xecute - one or more arguments (expressions) each optionally followed by a colon and another expression

As can be seen, there are few groupings and many single definitions. This made both the defining of the syntax initially and then implementing the grammar in the parser extremely difficult.

Re-entrant subroutines were created to parse expressions, entry references and MUMPS variables. It is possible for an entry reference parse to re-enter the entry reference code via a function parse for the namespace part of the initial entry reference; as it is possible for any of these three subroutines to call themselves or either of the other two.

The data structure used to contain the parsed routine is described on the next two pages. Note that both line source and scan codes are held in the structure. This is for debugging purposes and will be 'switched off' later.

PARSER DATA STRUCTURE

The Data Structure used by the MUMPS Parser is stored in a MUMPS on-disk multi-dimensional array.

For each routine (MUMPS for program or procedure or class) data is stored by line and indexed by line number; specifically ^%MUMPS(routine,line#) contains data for that line of the source code. For each line, the following fields are stored (All are optional):

Label	The routine line label
Formal Parameters	A list of formal parameters for a procedure
Commands	A list of all commands on the line indexed by command number

In MUMPS parlance, this looks like:

```
^%MUMPS(routine,line#) = Level number of this line (required)
^%MUMPS(routine,line#,"LABEL") = Line label if provided
^%MUMPS(routine,line#,"FP",n) = name of Formal Parameter n if provided
^%MUMPS(routine,line#,cmd#) contains all information for this command
```

Information stored for each command is:

Command word	The actual command
For index variable	MUMPSVAR (local) when the command is a For
Post Conditional	The EXPR comprising the post conditional Scan codes comprising the Post Conditional
Arguments	A list of scan codes for each of the arguments

The MUMPS specification looks like:

```
^%MUMPS(routine,line#,cmd#) = Command Word
^%MUMPS(routine,line#,cmd#,"FORPAR") = MUMPSVAR (local)
^%MUMPS(routine,line#,cmd#,"POST") = EXPR
^%MUMPS(routine,line#,cmd#,"POST",n) = Expression elements
^%MUMPS(routine,line#,cmd#,"POST","SC",n) = ScanCode
^%MUMPS(routine,line#,cmd#,"SC",n) = ScanCode
```

For debugging purposes, the line of source code is held in original form at:

```
^%MUMPS(routine,line#,0) = Line source
```

For each command, arguments are stored in the command data structure indexed by argument number as:

```
^%MUMPS(routine,line#,cmd#,Arg#) contains argument structure
```

This structure differs by command as follows:

```
BREAK   ,argno) = Value  (more than one argument is an error)
CLOSE   ,argno) = Value
DO      ,argno) = ENTRYREF (with PostConditional)
ELSE    -
FOR     ,argno) = Value for start value
             ,"I") = Value for incremental value
             ,"T") = Value for terminating value
GOTO    ,argno) = ENTRYREF (with PostConditional)
HALT    -
HANG    ,argno) = Value
IF      ,argno) = Value
JOB     ,argno) = ENTRYREF
KILL    ,argno) = MUMPSVAR
    or  ,argno) = SCLP
              ,n) = NAME value
LOCK    ,argno) = MUMPSVAR
    or  ,argno) = SCLP, SCPLUS or SCMINUS
              ,n) = MUMPSVAR
MERGE   ,argno) = MUMPSVAR (contains Source variable)
              ,"D") = MUMPSVAR (contains Destination variable)
NEW     ,argno) = NAME value or $ETRAP or $ESTACK
    or  ,argno) = SCLP
              ,n) = NAME value
OPEN    ,argno) = Value - channel no
              ,"p1") = Value - param 1
              ,"p2") = Value - param 2
              ,"to") = Value - timeout
```

```
                ,"ns") = Value - namespace
        QUIT    ,argno) = Value (more than one argument is an error)
        READ    ,argno) = Format (see Note 2) or [*] MUMPSVAR
        SET     ,argno) = Value (contains Source)
                ,"dn") = MUMPSVAR/function (Destination n - see Note 3)
        USE     ,argno) = Value - channel no
                ,"pn") = Value - param n -> n is 1 incremented by 1
                ,"ns") = Value - namespace
        VIEW    ,argno) = Value 1 (chan# always -1)
                ,"blk") = Value 2 (block#)
        WRITE   ,argno) Format (Note 2) or [*] Value (* as MOD)
        XECUTE  ,argno) = Value (incl postconditional if specified)

A Value is one of the following:
        1. expression           EXPR
        2. function             FUNCTION
        3. variable             MUMPSVAR
        4. literal              STRLIT, INTLIT or NUMBER=value

EXPR an expression is contained in a node as:
        NODE) = EXPR
            ,"POST") contains postcond Value if for XECUTE or $SELECT()
            ,n) contains an element which is one of:
                        scan code for operator (SCCAT -> SCNPAT) (<SCLP)
                        INTLIT, NUMBER or STRLIT '=' value
                        EXPR
                        SCINDIR (see note 1)
                        MUMPSVAR
                        FUNCTION

FUNCTION a system function or variable is held as:
        NODE) = FUNCTION
            ,"name") = full name/ENTRYREF of function/variable
            ,n) Contains argument n when "name" is NOT an ENTRYREF

MUMPSVAR a variable is held as:
        NODE) = MUMPSVAR
            ,n) contains Value for nth key
            ,"ind") Contains indirection
            ,"name") = name value
            ,"ssvn") = true if this is an ssvn
            ,"uci") (if defined)  0 = current or contains Value
            ,"MOD") = modifier (used for READ and WRITE)

ENTRYREF an entry ref is held as:
        NODE) = ENTRYREF
            ,n) Actuals = Value or '.' name for "by reference"
            ,"POST") contains postconditional Value if valid and specified
            ,"ind") Contains indirection
            ,"rou") = routine name
            ,"off") = offset
            ,"tag") = tag
            ,"uci") = uci name

Note 1: SCINDIR is an EXPR with the top node set to ISINDIR.

Note 2: Format arguments for WRITE and READ may consist of:
        1. string literal       STRLIT
        2. format char          SCOR (!) or SCMOD (#)
        3. '?' Value            SCPAT Value
        4. '/' name [ '(' Value { ',' Value } ')' ] - as for Local Var

Note 3: A Set Destination can be: A Variable (MUMPSVAR) or a FUNCTION
        One of: $EC[ODE], $ET[RAP], $K[EY] $X, $Y
        $E[XTRACT](Var[, Expr[, Expr]]), $P[IECE](Var[,Expr[, Expr[,Expr]]])
```

BUILT-IN FUNCTIONS

The language includes a number of built-in functions and system variables (functions with no arguments). These take a varying number and type of arguments as follows:

```
Functions/Variables - arguments are Value except where noted
Full Name        Abbr     Min      Max      Note     Arg1
$ASCII()         $A       1        2
$CHAR()          $C       1        no max
$DATA()          $D       1        1                 Variable
$EXTRACT()       $E       1        3        1
$FIND()          $F       2        3
$FNUMBER()       $FN      2        3
$GET()           $G       1        2                 Variable
$JUSTIFY()       $J       2        3
$LENGTH()        $L       1        2
$NAME()          $NA      1        2                 Variable
$ORDER()         $O       1        2                 Subscripted Variable
$PIECE()         $P       2        4        1
$QLENGTH()       $QL      1        1
$QSUBSCRIPT()    $QS      2        2
$QUERY()         $Q       1        2                 Variable
$RANDOM()        $R       1        1
$REVERSE()       $RE      1        1
$SELECT()        $S       1        no max            all args are Condition:Value
$STACK()         $ST      1        2
$TEXT()          $T       1        1                 Entryref
$TRANSLATE()     $TR      2        3
$VIEW()          $V       2        4

$DEVICE          $D
$ECODE           $EC                        1
$ESTACK          $ES                        2
$ETRAP           $ET                        1 and 2
$HOROLOG         $H
$IO              $I
$JOB             $J
$KEY             $K                         1
$PRINCIPAL       $P
$QUIT            $Q
$REFERENCE       $R
$STACK           $ST
$STORAGE         $S
$SYSTEM          $SY
$TEST            $T
$X               $X                         1
$Y               $Y                         1

Notes   1 = May be SET
        2 = May be NEWed
```

Also note that both $Piece() and $Extract() may be the destination for the *Set* command.

Definition of target (output code)

The nature of MUMPS requires that it has extensive runtime support. As examples the local symbol table is external to the compiled code and the database support code is required to run any MUMPS code.

As it is beyond the scope of this project to provide an extensive runtime system, it was decided to use the “object code” of the One Software 1999 implementation and to run with that runtime system to test the design of the compiler.

A copy of the definition of the output code is included on the following pages.

```
// File: mumps/include/opcodes.h
//
// module MUMPS header file - internal op codes (and only real opcodes)

/*      Copyright (c) 1999 - 2008
 *      Raymond Douglas Newman.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of Raymond Douglas Newman nor the names of the
 *    contributors may be used to endorse or promote products derived from
 *    this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
 * THE POSSIBILITY OF SUCH DAMAGE.
 *
 */


#ifndef _MUMPS_OPCODES_H_                   // only do this once
#define _MUMPS_OPCODES_H_

#define ENDLIN     0                        // End of line
#define OPHALT     1                        // Halt instruction
#define OPERROR    2                        // short -(ERR) follows
#define OPNOT      3                        // boolean (int) NOT
#define OPENDC     4                        // end of command
#define JMP0       5                        // jump if false
#define OPIFN      6                        // no arg if
#define OPIFA      7                        // if (check stack)
#define OPIFI      8                        // if indirect
#define OPELSE     9                        // else


#define OPADD      10                       // add top two on the stack
#define OPSUB      11                       // (sp-2) - (sp-1)
#define OPMUL      12                       // mult top two on the stack
#define OPDIV      13                       // (sp-2) / (sp-1)
#define OPINT      14                       // integer divide (MUMPS style)
#define OPMOD      15                       // modulus (MUMPS style)
#define OPPOW      16                       // x to the power y
#define OPCAT      17                       // a concatenated with b
#define OPPLUS     18                       // unary plus
#define OPMINUS    19                       // unary minus

#define OPEQL      20                       // a = b
#define OPLES      21                       // a < b
#define OPGTR      22                       // a > b
#define OPAND      23                       // a & b
#define OPIOR      24                       // a ! b (inclusive or)
#define OPCON      25                       // a contains b
#define OPFOL      26                       // a follows b
#define OPSAF      27                       // a sorts after b
#define OPPAT      28                       // a pattern matches b
```

```
#define OPHANG     29                    // hang

#define OPNEQL     30                    // not a = b
#define OPNLES     31                    // not a < b
#define OPNGTR     32                    // not a > b
#define OPNAND     33                    // not a & b
#define OPNIOR     34                    // not a ! b (inclusive or)
#define OPNCON     35                    // not a contains b
#define OPNFOL     36                    // not a follows b
#define OPNSAF     37                    // not a sorts after b
#define OPNPAT     38                    // not a pattern matches b
//spare   39

//spare   40
#define CMSET      41                    // set
#define CMSETE     42                    // set $E()
#define CMSETP     43                    // set $P()
#define OPNAKED    44                    // set NAKED from mvar on astk
#define CMFLUSH    45                    // flush inputs
#define CMREADS    46                    // read star
#define CMREADST   47                    // read star with timeout
#define CMREAD     48                    // read variable
#define CMREADT    49                    // read variable t/o

#define CMREADC    50                    // read variable count
#define CMREADCT   51                    // read variable count, t/o
#define CMWRTST    52                    // write star
#define CMWRTNL    53                    // write !
#define CMWRTFF    54                    // write #
#define CMWRTAB    55                    // write ?expr
#define CMWRTEX    56                    // write expression
#define CMUSE      57                    // use (args) ch, a1, a2, ...
#define CMOPEN     58                    // open chan, p1, p2, timeout
#define CMCLOSE    59                    // close channel

#define OPSTR      60                    // string follows in line
#define OPVAR      61                    // eval var name follows
#define OPMVAR     62                    // build mvar, name follows
#define OPMVARN    63                    // build mvar, (null OK)
#define OPMVARF    64                    // bld mvar, no null, full size
#define INDEVAL    65                    // eval name indirection
#define INDMVAR    66                    // mvar name indirection
#define INDMVARN   67                    // mvar name ind (null ok)
#define INDMVARF   68                    // mvar name ind, full size
//spare   69

#define OPBRK0     70                    // argless break
#define OPBRKN     71                    // break with arguments
#define OPDUPASP   72                    // duplicate top of astk
//spare   73 -> 79

#define VARD       80                    // $D[EVICE]
#define VAREC      81                    // $EC[ODE]
#define VARES      82                    // $ES[TACK]
#define VARET      83                    // $ET[RAP]
#define VARH       84                    // $H[OROLOG]
#define VARI       85                    // $I[O]
#define VARJ       86                    // $J[OB]
#define VARK       87                    // $K[EY]
#define VARP       88                    // $P[RINCIPAL]
#define VARQ       89                    // $Q[UIT]

#define VARR       90                    // $R[EFERENCE]
#define VARS       91                    // $S[TORAGE]
#define VARST      92                    // $ST[ACK]
#define VARSY      93                    // $SY[STEM]
#define VART       94                    // $T[EST]
#define VARX       95                    // $X
#define VARY       96                    // $Y
//spare   97 -> 99
```

```
#define FUNA1    100                    // $A[SCII] 1 arg
#define FUNA2    101                    // $A[SCII] 2 arg
#define FUNC     102                    // $C[HARACTER]
#define FUND     103                    // $D[ATA]
#define FUNE1    104                    // $E[XTRACT] 1 arg
#define FUNE2    105                    // $E[XTRACT] 2 arg
#define FUNE3    106                    // $E[XTRACT] 3 arg
#define FUNF2    107                    // $F[IND] 2 arg
#define FUNF3    108                    // $F[IND] 3 arg
#define FUNFN2   109                    // $FN[UMBER] 2 arg

#define FUNFN3   110                    // $FN[UMBER] 2 arg
#define FUNG1    111                    // $G[ET] 1 arg
#define FUNG2    112                    // $G[ET] 2 arg
#define FUNJ2    113                    // $J[USTIFY] 2 arg
#define FUNJ3    114                    // $J[USTIFY] 3 arg
#define FUNL1    115                    // $L[ENGTH] 1 arg
#define FUNL2    116                    // $L[ENGTH] 2 arg
#define FUNNA1   117                    // $NA[ME] 1 arg
#define FUNNA2   118                    // $NA[ME] 1 arg
#define FUNO1    119                    // $O[RDER] 1 arg

#define FUNO2    120                    // $O[RDER] 1 arg
#define FUNP2    121                    // $P[IECE] 2 arg
#define FUNP3    122                    // $P[IECE] 3 arg
#define FUNP4    123                    // $P[IECE] 4 arg
#define FUNQL    124                    // $QL[ENGTH]
#define FUNQS    125                    // $QS[UBSCRIPT]
#define FUNQ1    126                    // $Q[UERY] 1 arg
#define FUNQ2    127                    // $Q[UERY] 2 arg
#define FUNR     128                    // $R[ANDOM]
#define FUNRE    129                    // $RE[VERSE]

#define FUNST1   130                    // $ST[ACK]
#define FUNST2   131                    // $ST[ACK] 2 arg
#define FUNT     132                    // $T[EXT]
#define FUNTR2   133                    // $TR[ANSLATE] 2 arg
#define FUNTR3   134                    // $TR[ANSLATE] 3 arg
#define FUNV2    135                    // $V[IEW] - 2 arg
#define FUNV3    136                    // $V[IEW] - 3 arg
#define FUNV4    137                    // $V[IEW] - 4 arg
#define CMVIEW   138                    // VIEW command - 4 args
#define CMMERGE  139                    // merge 1 variable from nxt

#define CMDOWRT  140                    // DO from WRITE /xxx[(param)]
#define CMDOTAG  141                    // DO tag in this rou [args]
#define CMDOROU  142                    // DO routine (no tag) [args]
#define CMDORT   143                    // DO routine, tag [args]
#define CMDORTO  144                    // DO routine, tag, off [args]
#define CMDON    145                    // DO - no arguments
#define CMJOBTAG 146                    // JOB tag in this rou [args]
#define CMJOBROU 147                    // JOB routine (no tag) [args]
#define CMJOBRT  148                    // JOB routine, tag [args]
#define CMJOBRTO 149                    // JOB routine, tag, off [args]

#define CMGOTAG  150                    // GOTO tag in this rou
#define CMGOROU  151                    // GOTO routine (no tag)
#define CMGORT   152                    // GOTO routine, tag
#define CMGORTO  153                    // GOTO routine, tag, off
#define CMXECUT  154                    // XECUTE
#define CMXECI   155                    // XECUTE indirect
#define CHKDOTS  156                    // check current level
#define CMQUIT   157                    // QUIT - no arg (not FOR)
#define CMQUITA  158                    // QUIT with argument
//spare   159

#defineCMLCKU    160                    // un LOCK all
#define CMLCK    161                    // LOCK #args()
#define CMLCKP   162                    // LOCK + #args()
```

```
#define CMLCKM   163                    // LOCK - #args()
#define CMNEW    164                    // NEW
#define CMNEWB   165                    // NEW #args() - new except
#define CMKILL   166                    // kill 1 variable
#define CMKILLB  167                    // kill but() args
#define NEWBREF  168                    // push null for NEW by ref
#define VARUNDF  169                    // point at VAR_UNDEFINED

#define LINENUM  170                    // set rou line number
#define LOADARG  171                    // load args (illegal in line)
#define JMP      172                    // unconditional jump
#define CMFOR0   173                    // argless FOR
#define CMFOR1   174                    // FOR with 1 argument
#define CMFOR2   175                    // FOR with 2 arguments
#define CMFOR3   176                    // FOR with 3 arguments
#define CMFORSET 177                    // setup FOR
#define CMFOREND 178                    // Jump to end of line
#define OPNOP    179                    // NOP

#define INDREST  180                    // restore isp & mumpspc
#define INDCLOS  181                    // CLOSE arg indir
#define INDDO    182                    // DO arg indir
#define INDGO    183                    // GOTO arg indir
#define INDHANG  184                    // HANG arg indir
#define INDIF    185                    // IF arg indir
#define INDJOB   186                    // JOB arg indir
#define INDKILL  187                    // KILL arg indir
#define INDLOCK  188                    // LOCK arg indir
#define INDMERG  189                    // MERGE arg indir

#define INDNEW   190                    // NEW arg indir
#define INDOPEN  191                    // OPEN arg indir
#define INDREAD  192                    // READ arg indir
#define INDSET   193                    // SET arg indir
#define INDUSE   194                    // USE arg indir
#define INDWRIT  195                    // WRITE arg indir
#define INDXEC   196                    // XECUTE arg indir
//spare   197 -> 199

//spare   200 -> 229

//spare   230 -> 233
#define XCWAIT   234                    // Xcall $&%WAIT()
#define XCCOMP   235                    // Xcall $&%COMPRESS()
#define XCSIG    236                    // Xcall $&%SIGNAL()
#define XCHOST   237                    // Xcall $&%HOST()
#define XCFILE   238                    // Xcall $&%FILE()
#define XCDEBUG  239                    // Xcall $&DEBUG()

#define XCDIR    240                    // Xcall $&%DIRECTORY()
#define XCERR    241                    // Xcall $&%ERRMSG()
#define XCOPC    242                    // Xcall $&%OPCOM()
#define XCSPA    243                    // Xcall $&%SPAWN()
#define XCVER    244                    // Xcall $&%VERSION()
#define XCZWR    245                    // Xcall $&%ZWRITE()
#define XCE      246                    // Xcall $&E()
#define XCPAS    247                    // Xcall $&PASCHK()
#define XCV      248                    // Xcall $&V()
#define XCX      249                    // Xcall $&X()

#define XCXRSM   250                    // Xcall $&XRSM()
#define XCSETENV 251                    // Xcall $&%SETENV()
#define XCGETENV 252                    // Xcall $&%GETENV()
#define XCROUCHK 253                    // Xcall $&%ROUCHK()
#define XCFORK   254                    // Xcall $&%FORK()
#define XCIC     255                    // Xcall $&%IC()

#endif                                  // _MUMPS_OPCODES_H_
```

Conclusion

The purpose of this project was to start to re-implement the MUMPS language compiler properly with the main thrust being a full analysis of the syntax and semantics of the language. This was to have been done before starting coding but it was found that a certain amount of prototyping assisted in the design.

We have described MUMPS using EBNF Grammar. A set of scan codes has been defined and the scanner implemented to produce the scan code stream.

The parser data structure has been designed and the parser written to populate the data structure in two passes. The first pass breaks each line into a label and discrete commands; the arguments are left unparsed.

The second pass parses the arguments; this being different for each command.

The next step is to apply further analysis to the parsed database to apply any possible optimisation and to then produce output for the MUMPS V1 environment.

Should this show a reasonable increase in speed, I will then attempt to re-code the scanner and parser in c to incorporate it into the current environment.

Although using tools like LEX and YACC proved to be not feasible, a proper design has produced a relatively clean output from the parser. The project has also shown that the MUMPS language itself is suited to compiler writing.

Ray Newman
13 November 2008