MUMPS 1983 - A SECOND LOOK

Charles S. Volkstorf

CompuMedical Inc.
Boston, MA

In 1977 the Mumps Development Committee (MDC) completed the original specifications for Standard Mumps, which was accepted as a national standard by the American National Standards Institute (ANSI), titled ANSI X11.1 - 1977. On September 15, 1983, the MDC completed its proposed five-year revision to Mumps, titled Draft ANSI X11.1 - 1983. A copy of this document was forwarded to each member of their canvass list, including this author, for a formal vote. This article describes and analyzes the proposed revision to Standard Mumps. We also discuss possible enhancements to Mumps and its implementations.

I. THE CHANGES FROM 1977 MUMPS TO 1983 MUMPS

The MDC took polls in both 1977 [1] and 1980 [2] to determine what language features were most needed in Mumps. In both polls, the top concern was Parameter Passing: the ability to DO a subroutine and reset only its output variables, automatically avoiding variable name conflicts. This capability is almost universally available in other programming languages, and is in fact the biggest deficiency in Mumps. It makes invoking existing routines from new routines more difficult, time-consuming, and error prone. Close behind Parameter Passing in both polls is Block Structuring: the ability to structure code into blocks to make it easier to understand. Both Parameter Passing and Block Structuring are missing from 1983 Mumps.

Altogether the MDC made 14 changes to the language. Ten of these correct quirks, shorten code, or provide conveniences. The other four changes are significant additions to the language: $ORDER function, SET $PIECE, JOB command, and Fixed-length READ. The latter two are fairly well defined, and will not be discussed here.

Addition 1. $ORDER function

Subscripts in 1977 Mumps are limited to non-negative integers. When we $NEXT through a file (or local array), the value -1 is returned when we pass the last subscript. However, implementors of Mumps now want to allow subscripts to be strings. If this includes the string "-1", then when $NEXT returns -1, we have trouble telling if we've reached subscript -1 or the end of the subscripts. As the MDC states it [3, p. I-20], "$N will return ambiguous results for...negative numeric subscript values." Consequently, the MDC introduced a new function, $ORDER, to go through files, which uses the empty string "" as the end-of-file indicator instead of -1, and disallowed "" as a subscript.

This approach confuses the issue. Programmers are unsure as to which to use, $NEXT or $ORDER. $NEXT is used extensively in 1977 Mumps and is included in 1983 Mumps, but now the MDC states [3, p. I-19], "The use of the $ORDER function is strongly encouraged in place of $NEXT." $NEXT is discarded as deadwood in the language.

AN ALTERNATE APPROACH

A simpler, and more effective, approach would be to use $NEXT with string subscripts, and not have to introduce a new function. How to best do this depends on which strings we actually allow as subscripts. Here the MDC is inconsistent. The definition of $NEXT [3, p. I-20] refers to "arrays which have negative numeric subscript values", for which $ORDER was introduced. The definition of allowable subscripts [3, p. III-2], however, states, "when...numeric...it is further subject to the restriction of...nonnegative numbers only." A notice on page 2, stating, "NOTE: In cases of conflict between Part I and Part II, Part II will defer to Part I", is of no help, as this inconsistency occurs in parts I and III.

If we disallow negative subscripts, we can continue to use $NEXT as is. This is practical because negative subscripts are virtually never used and in fact are not necessary. In 11 years of Mumps programming, including 6 years of string subscripts, this author has never needed or encountered them. If ever needed, we can easily avoid them by adding an offset, such as 1E6 (that is, 1000000) to convert any negative number into a positive one, before using it as a subscript. The MDC already disallows strings with a length greater than 31 characters or containing control characters from being subscripts [3, p. III-2]. This is reasonable and doesn't actually reduce

the power or ease of use of Mumps. Negative subscripts could likewise be disallowed.

Solution 1. Disallow negative subscripts. Use $NEXT as is.

The subscript collating order could also be better defined, without disrupting existing (1977 Mumps) programs. In 1983 Mumps, the collating order is: empty string, numbers, non-numeric strings. Without $ORDER, we could let the empty string collate in its natural order, with the other non-numeric strings. The collating order would simplify to: numbers, non-numeric strings.

We could also allow the empty string to be a subscript. It is supported by the popular B-tree, and is disallowed in current implementations only as a convention for the sake of $ORDER. This would occasionally be useful, in tallying the frequency of non-numeric field values, as in ^STAT(VALUE)=COUNT. Currently, VALUE="" must be checked for separately, which is sometimes overlooked until a subscript error occurs.

(Actually, $NEXT isn't ambiguous. When $NEXT returns -1, it is subscript -1 when the previous subscript collates before -1 and -1 is $DEFINED. Otherwise, it is the end-of-file. Thus $ORDER isn't really necessary at all.)

DECLARING THE END-OF-FILE

If we decide that we must have negative subscripts, we can still continue to use $NEXT. As an alternative to the simple check described above, we could declare the value that $NEXT returns when it reaches the end-of-file. The default would be -1, so that existing programs would continue to work. We could change it to "" with the command NEXT "" and back to -1 with the command NEXT -1. Alternately, we could use the commands YEAR 1983 and YEAR 1977. The programmer could begin his system with the declaration NEXT "" and use $NEXT throughout, instead of introducing $ORDER.

Solution 2. Declare with a new command (NEXT or YEAR) that the $NEXT end-of-file indicator is "" instead of -1.

USING A SYSTEM VARIABLE

Another solution is to detect the end-of-file with a System Variable. $DEFINED with no arguments could equal 0 if the end-of-file is reached, and 1 otherwise. Then, after we SET SUB=$NEXT(^(SUB)) we QUIT:'$DEFINED.

We could extend this further by having the $DEFINED variable return the $DEFINED function of the last value returned by $NEXT, and 0 for end-of-file. We occasionally check this after we $NEXT. The compiler could determine this value only when the $DEFINED variable is actually referenced. Also, there is low overhead in evaluating it under the popular B-tree file structure.

Solution 3. Let $NEXT set a $DEFINED System Variable equal to 0 or 1 to indicate the end-of-file (or the $DEFINED function of the next subscript). Then we QUIT:'$DEFINED to end a $NEXT loop.

THE FOUR SOLUTIONS

In summary, we have developed three solutions to the $NEXT problem, as alternatives to the $ORDER proposal given in 1983 Mumps.

1. No negative subscripts, $NEXT as is.
2. NEXT or YEAR command to change the $NEXT end-of-file indicator.
3. $DEFINED System Variable to detect the end-of-file.
4. $ORDER to replace $NEXT.

Which is preferable depends on whether or not negative subscripts are allowed. The MDC is inconsistent on the question of negative subscripts. Our analysis indicates that negative subscripts are unnecessary, and that using $NEXT as is (number 1) offers the best solution because it is simple, effective, and requires the minimal change to Mumps.

Addition 2. SET $PIECE

SET $PIECE allows us to change a single field within a string of delimited fields. For example, if NODE equals NAME_"^"_DR_"^"_DEPT then SET $PIECE(NODE,"^",3)=NEW resets DEPT within NODE to NEW. A similar syntax, using $EXTRACT instead of $PIECE, has also been suggested.

SET $PIECE is useful, but has a syntax and semantics foreign to Mumps. Select functions are allowed on the left side of the equals in the SET command. One argument to these functions is altered when it is executed. This occurs nowhere else in Mumps.

$PIECE-5

The capabilities of SET $PIECE may be provided within the existing structure of Mumps. A fifth argument to the $PIECE function could indicate the new value to replace the four-argument interval of pieces, and $PIECE itself would return the result. Then instead of SET $PIECE(NODE,"^",3)=NEW we would SET NODE=$PIECE(NODE,"^",3,3,NEW). We replace "^" pieces 3 through 3 in NODE with NEW. $EXTRACT could likewise be extended. This would be consistent with the syntax and semantics of 1977 Mumps. We will call this proposal $PIECE-5.

- SET $PIECE is more self-descriptive than $PIECE-5, but this is because SET $PIECE is in the Cobol style of having an arbitrary sequence of reserved words and expressions for each construct. $PIECE-5 is in the Mumps style of having a small set of well-chosen functions and operators.

- $PIECE-5 provides more general capabilities than SET $PIECE. SET $PIECE, being a syntactic aberration, can only be used as the argument of a SET command. $PIECE-5, being a normal Mumps function, may also be used in any other context in which an expression may appear. Furthermore, by adding a fifth argument to $EXTRACT to indicate an alternate "filler character", $PIECE-5 allows capabilities not possible under SET $PIECE at all. The representative from Mass General Hospital (where Mumps originated) identified two uses for alternate padding values [4]: "a string of dashes" (by padding with "-"), and "a string of alternate dashes and spaces" (by padding with "- ").

- $PIECE-5 is generally more efficient disk-wise than SET $PIECE. For example, suppose that NODE is stored in ^PAT(ID). Then the shortest and most common way to reset ^PAT would be as follows.

Under SET $PIECE:
```
SET $PIECE(^PAT(ID),"^",3)=NEW
```

Under $PIECE-5:
```
SET ^PAT(ID)=$PIECE(NODE,"^",3,3,NEW)
```

The SET $PIECE code retrieves ^PAT(ID), resets it, then files it away. We reference ^PAT(ID) twice. The $PIECE-5 code calculates the new value of ^PAT(ID) before retrieving it, and sets it in directly. ^PAT(ID) is referenced only once.

While the increased readability of SET $PIECE is desirable, the loss in capabilities and efficiency makes $PIECE-5 preferable.

TRANSLATING UNDEFINED VALUES INTO THE EMPTY STRING

SET $PIECE also stipulates that if the first argument is not defined, an empty string is used for its value. This concept is useful in tallying statistics, in which a total of 0 for a given value is left undefined. However, SET $PIECE does not increment a counter. If we SET $PIECE(^PAT(ID),"^",3)=DEPT and ^PAT(ID) is not defined, it will create a string with the indicated department, but with a null patient name and doctor. It should abort and be fixed, not continue to process and be discovered later as a mysterious entry in the file.

FURTHER ANALYSIS

Another proposal centering around $PIECE and debated at length by the MDC is the PIECED VARIABLE. Under PIECED VARIABLE, the expression NODE$3 would have been equivalent to $PIECE(NODE,"^",3) after we SET $P="^". The advantage to PIECED VARIABLE is that it would shorten the code required to retrieve from the file. However, if it is brevity that we seek, we could let &EXPR represent the "ultimate naked", to the last executed function, local array, or file reference. This would be even shorter than both PIECED VARIABLE and file naked references, would also include $EXTRACT, give us a naked to local arrays, and various other capabilities. The vertical bar character could be substituted for "&" to be more distinct. A compiler could avoid remembering the last executed reference if it weren't followed by a reference to this new unary operator.

PIECED VARIABLE is attractive because $PIECE is used frequently, often with the same first and second arguments. In general, then, we are attempting to eliminate duplicate Mumps code.

The original process of concatenating multiple fields, with a delimiter inbetween, could be made less repetitive with a new function, such as $KONCATENATE. The first argument would be the delimiter, with an arbitrary number of successive arguments for fields. Then we could
SET ^PAT(ID)=$KONCATENATE("^",NAME,DR,DEPT).

PIECED VARIABLE is shorter than $PIECE, but there are still repeated references to the variable. Rather than shrinking each individual reference, we could eliminate all repetition by combining these references into one. Then we would be manipulating a list of values, which mathematicians call a TUPLE.

TUPLES

We need to indicate multiple piece numbers and multiple variables in one argument to the SET command. The character "$" would serve as a separator in both cases, to provide:

```
SET NAME$DR$DEPT=$PIECE(NODE,"^",1$2$3)
```

It would also be useful to allow references to $EXTRACT, local arrays, and files, to allow ranges and conditions, to go through TUPLES with the FOR command, and to write out TUPLES with the WRITE command. This would be relatively easy to define, but powerful and efficient.

A TUPLE is either:

1. A single variable or expression.
2. A TUPLE followed by "$" and another TUPLE.
3. Any of the following, with a TUPLE or [EXPR1]:[EXPR2][:TVEXPR] as the last argument/subscript: $PIECE with 3 arguments, $EXTRACT with 2 arguments, local array reference, or a file reference. EXPR1, EXPR2, and TVEXPR are the minimum, maximum, and condition, respectively, and each is optional.

A SET command (or possibly a new command such as TUPLE) would assign a tuple of values to an equal number of variable names, or to a local array or a file using the last argument/subscripts in the value as the new subscript.

This provides a wide range of capabilities under various file designs. For example:

1. Set file node pieces into individual variables:
   SET NAMEDRDEPT=$PIECE(NODE,"^",1$2$3)

2. Save individual variables into a file:
   SET ^PAT(ID,1$2$3)=NAMEDRDEPT

3. Set file node pieces into a local array:
   SET P=$PIECE(NODE,"^",:)

4. Move file nodes into a local array:
   SET P=^PAT(ID,:)

5. Save a local array into a file:
   SET ^PAT(ID)=P(:)

6. Move data from one file to another:
   SET ^PAT(ID)=^TEST(NUM,:)

7. Go through a file:
   FOR NODE=^PAT(ID,:)

8. Go through multiple levels at once:
   FOR ID$DATE$CHARGE=^PAT(:,:,:)

9. Go through a local array:
   FOR NODE=P(:)

10. Go through pieces in a node from a file:
    FOR VALUE=$PIECE(NODE,"^",:)

11. Query Processing:
    All patients who saw doctor DR and had charge CHARGE:
    FOR ID=^PAT(::$PIECE($NODE,"^",2)=DR,CHARGE)
    where $NODE is the value stored at that subscript.

12. Page a file into a local array repeatedly:
    SET P=^PAT(ID,LAST::$STORAGE>1000)

13. Examine our local symbol table one CRT screen's-worth at a time:
    WRITE A:E

14. Examine individual local arrays:
    WRITE ID,SPECIMEN(:),TEST(:)

Programmers could then manipulate lists of local variables, such as NAME, DR, and DEPT, as easily as they manipulate local array references, such as P(1), P(2), and P(3). Individual local variable names could be used instead of local arrays, contributing to the readability of Mumps programs. As an argument to a FOR command, TUPLES provide another solution to the $NEXT/$ORDER problem. Similar capabilities exist with $KONCATENATE. Properly defined, $KONCATENATE encompasses the MDC's SET $PIECE. In general, S $P(A,B,C,D)=E may be represented by S A=$K(B,$P(A,B,:C-1),E,$P(A,B,D+1:)), although in this case it does lack the clarity of SET $PIECE.

SURVEYING LOOPS

Are there other sets that would make useful TUPLE values? The values we have mentioned so far - $PIECE, $EXTRACT, local arrays, and files - are all sets that Mumps programmers often loop through. Thus we can systematically determine good candidates for TUPLES by surveying loops in existing Mumps programs.

The question of designing good loop structures for a programming language was pondered by the ALGOL committee over 15 years ago. In his famous letter to the editor of Communications of the ACM (March, 1968), "GOTO Statement Considered Harmful", Edgar Dijkstra deplored the use of the GOTO statement, while helping to develop the Algol WHILE construct:

WHILE (condition) DO (statements) OD

This gave birth to the modern concept of "Structured Programming". The WHILE command continues to be the basic looping construct in popular "structured" languages, such as Algol and Pascal. While this is a very general capability, it doesn't codify the actual processes that programmers need to perform. The initializing, incrementing, and checking are still left up to the programmer. It only replaces the GOTO. The TUPLE construct, in contrast, performs all of these chores for the programmer. You merely indicate what data structures you would like to traverse.

To determine what capabilities are actually needed, this author wrote a Mumps program to detect and list every loop in a given Mumps program. It was run against two major systems. The results were as follows.

Number of routines: 194
Number of characters of code: 414,365
Number of loops: 2,038
Frequency of loops: one per 203.3 characters of code.

A representative sample of 569 of these loops was manually classified. A total of 10 types of loops were detected.

1. READ (221=38%)
   A value is read from the terminal, processed, and then we GOTO the read for another input.
2. FILE $NEXT (136=24%)
   We loop through a file subscript.
3. LIST (76=13%)
   A sequential list is traversed.
4. LOCAL ARRAY $NEXT (68=12%)
   We loop through a local array subscript.
5. FIXED SET (17=3%)
   We process each value in a fixed set, such as 1,3, and 10.
6. PIECES (15=3%)
   We examine each $PIECE of a given string.

7. SEQUENTIAL SEARCH (14=3%)
   We increment a value until a certain condition occurs.
8. FIXED MATH INTERVAL (10=2%)
   We examine a fixed range of integers.
9. FIELD FROM/TO (7=1%)
   We examine all integers in a variable from/to range.
10. CHARACTERS (5=1%)
    Each character in a string is examined.

The FOR in Mumps easily handles the 26% LIST, FIXED SET, PIECES, SEQUENTIAL SEARCH, FIXED MATH INTERVAL, FIELD FROM/TO, and CHARACTERS loops. The TUPLE proposal handles the 36% FILE $NEXT and LOCAL ARRAY $NEXT (also simplifying PIECES and CHARACTERS). How about the remaining 38% READ loops?

ARGUMENTLESS FOR

A READ loop performs no initializing or incrementing, only a check to see if more input is entered. However, a quit can be used to check for a condition to terminate a FOR, so all we really need is a way to repeat a process (without using GOTO). Thus we need a FOR that has no argument. This fits very well into Mumps' syntax. Many commands can be argumentless, and it is a natural syntax for these semantics. Then a READ loop could be expressed as
FOR DO READ QUIT:INPUT="". In fact, any of these loops might be coded using the argumentless FOR, by initializing and then performing the necessary incrementing and checking within its scope.

II. ADDITIONAL ENHANCEMENTS

We now consider enhancements to popular implementations of Mumps. While these capabilities fall outside the definition of the language per se, we feel that they would increase the effectiveness of using Mumps.

When the machine fails (crashes), save all relevant information into a block or file. Save the buffer pool contents and/or block numbers to the console, a block, a file, and/or their appropriate blocks when it crashes and/or when we initiate an Operating System subroutine through the console switches, in order to quickly detect/correct any file corruption being caused by the crash.

Have hardware diagnostics on the Mumps pack, bootable and/or callable under Mumps for periodic automatic Preventative Maintenance.

Store integer subscripts (72% of all subscripts, by my own survey) two digits per byte rather than just one. Convert existing files with a machine language program, or allow both formats to exist.

Multi-task disk sets. SET ^G(A,B)=C should process on its own while the program initiating it continues (unless/until it reaches an affected reference to ^G before it is completed).

Sort the Disk Queue by the cylinder referenced, to process the whole Queue in one clean sweep of the disk rather than going back and forth at random. Seek time is the vast majority of disk time.

Process one disk queue per disk drive, not just the single Disk I/O Queue per CPU displayed in the System Status utility. This would allow simultaneous execution of disk drives rather than their idly waiting for each other.

Totally compile the 99.4% (by my own survey) of Mumps that can be, and interpret execution and indirection (usually, and advisably, avoidable at that), rather than semi-compiling 100% of it.

(See the prior issue of the MUG Quarterly for additional possibilities.)

III. SUMMARY AND CONCLUSIONS

A thorough examination of 1977 Mumps, 1983 Mumps, and the deliberations of the MDC, shows the following.

1. The top concerns of vendors and users, Parameter Passing and Block Structuring, are still missing from 1983 Mumps.

2. There are four significant additions made to Mumps. Alternate approaches, that are simpler, more general, and more efficient, should be considered.

3. The document itself (Draft ANSI X11.1 - 1983) should be examined closely for inconsistencies and erroneous definitions. Examples of each Mumps construct and the intended result should be given, to rectify ambiguities and implementors' misinterpretations.

In conclusion, we feel that it is clear that the proposed revision to Mumps should be withdrawn and further studied.

ACKNOWLEDGEMENTS

I would like to thank Mumps experts Bob Craig, Michael Ginsberg, Andrew Levine, and Alan Whitney for proof-reading this paper and suggesting numerous improvements.

REFERENCES

1. MDC (1977) Meeting #13, Boston, Massachusetts, March 24, 1977.

2. MDC (1980) Meeting #21, Washington D.C., October 30, 1980.

3. MDC (1983) Draft ANSI X11.1 American National Standard MUMPS

4. MDC (1979) Meeting #19, McLean, Virginia, October 18, 1979.