Extending MUMPS to Provide Better Program Development Capabilities

C.S. Volkstorf
CompuMedical Inc.
Boston, Massachusetts

Standard MUMPS [1] is now undergoing its five-year ANSI revision [2]. While the debate over language features continues [3], the issues concerning the efficient development and maintenance of MUMPS programs remains outside the scope of Standard MUMPS. MUMPS contains no provisions for the creation or editing of programs themselves. Providing this capability is left up to individual implementors. There is, however, a de facto standard for programming aids due to the preponderance of certain extensions (Z commands) within popular implementations, as follows.

PRINT or ZPRINT Display one or more lines of a given program.
ZSAVE Create a program.
ZLOAD Declare the "current" program.
ZREMOVE Delete one or more lines from the current program.
ZINSERT Add a line to the current program.

The purpose of this note is to examine these popular extensions to MUMPS, and to suggest other possibilities that, we feel, would make program development and maintenance easier, more efficient, and less error prone.

Our first suggestion is to interchange the definitions of "routine" and "program" to be consistent with common usage of these terms. We use that terminology throughout this article.

Program Development

Print. As programmers develop, test, and debug a MUMPS program, each time it aborts on an error, the code surrounding the error is examined in order to discover and fix the problem. However, the lines above the line, TAG, at which the error occurred cannot be directly addressed. If we print the program from the top on down, using P +1:TAG, we lose the output on the screen, such as a report display showing where execution was, variable values that we WRITE, and the line containing the error. All this scrolls off the screen. We need to be able to P TAG-5:TAG+5. (It is said that the manner in which MUMPS programs are commonly stored prohibits stepping backwards from a given line, but this can be accomplished by maintaining two pointers to lines as we step through the program, the second pointer 5 lines behind the first, in this case.) Alternately, list the line number, say 50, when a program aborts, and we could print using P +45:+55. This is also needed when the programmer wishes to examine a line in its "context".

Another similar approach is to define the "current" line as the last line printed or following the last line erased. Then P * prints the current line, and P + prints the next line, allowing us to step through a program. Also, P *5 prints the next 5 lines and P TAG*5 prints 5 lines, starting at line TAG.

Another useful approach is to define a line's "subroutine" as the sequence of lines from the prior blank line to the following blank line. Then P *TAG prints its subroutine, P ** prints the current subroutine, and P #N prints the Nth subroutine. This would also encourage modular programming by delineating subroutines with blank lines. We could also leave out the P and blank.

ZREMOVE. ZREMOVE (I call it ZRASE) could have the same arguments as PRINT, to erase lines prior to tags, the current line, a tag's subroutine, the current subroutine, and the Nth subroutine. ZR + should be disallowed, however, since it is not useful and can be easily typed by accident. ZR TAG should abort when line TAG doesn't exist, rather than processing normally, to notify the programmer of typos or other causes of mistaken input. Similarly, ZR AAA:BBB should abort when BBB does not exist, rather than erasing AAA and all lines following it (without warning). If these lines are erased, a subsequent ZSAVE loses the lines permanently. On the other hand, if the code is reloaded, we lose the changes made since the last executed ZSAVE. Also, if AAA does not exist, this command is ignored. This too should abort.

Another suggestion is to let ZR (AAA:BBB) perform an "exclusive erase", for example, to leave lines AAA through BBB to be segmented into a new program.

Many of these ideas could be incorporated into a program editor, although that would be less efficient than providing them as MUMPS commands, as well as add the overhead of entering and leaving the editor each time they are to be used.

Editor. The program editor itself should be written in machine language, for example, accessible by a ZEDIT command, rather than in MUMPS. Features of MUMPS have forced implementors to resort to interpreters or semi-compilers instead of compilers (also see [3]). Furthermore, benchmarks have shown the Xecute command and indirection to actually be significantly slower under semi-compilers than under interpreters, further detracting from the performance of MUMPS-based editors. Since the editor is written only once, and is not altered by individual users, the overhead of interpreting or semi-compiling can be avoided by providing it in machine language. This is especially desirable since programmers use the editor almost continually in program development and maintenance.

Tags. Duplicate tags should not be allowed. They are not needed, cause mysterious bugs if undetected, and are difficult to correct when detected because all tag references refer to the first occurrence. Also, if ZR TAG or ZR TAG+N is used to mean the second occurrence, the wrong line is erased.

Typographical error detection. Typographical errors should abort the whole command rather than executing up to the illegal character and then aborting with a <SYNTX> error. For example, K ^PAT<ID) executes K ^PAT then aborts. ZS PR*G overwrites program PR then aborts. This is also true if there is an invisible control character in place of "*". Similarly, if ZL PRG is used to see if program name PRG is in use, (with an accidental control character between R and G) a <NOPGM> is displayed, and we think that we can safely use PRG, when in fact PR was being checked. Syntax errors during user sign-on also add mysterious <SYNTX> errors to the Error Monitor after the program runs to completion.

The RSTS/RSX file protection scheme is inappropriate for MUMPS's hierarchical file structure, and does not solve this problem. We need to be able to prohibit K ^PAT but allow K ^PAT(ID) to delete individual data fields. We could also prevent still others of the aforementioned problems with typographical errors if we could put protection codes on programs as well.

Typos in deleting programs (by ZR ZS PROG) could be avoided by ZS - (ZS space hyphen) deleting the current program, after we visually ascertain it is the one we wish to delete. If a program is still accidentally deleted or overwritten, it could be recovered if the Operating System kept a copy of the last program deleted or overwritten, and loaded it with ZL *

ZSAVE. Programmers often create a small, temporary, unfiled program in their local partition area, then expand it until it is large enough to be worthwhile to ZSAVE, or to create a new production program. However, if they call another program, for example, to perform a Global List or determine the time, they lose what they were working on. The program contained in a partition should be saved and restored, even if it hasn't been given a program name (using ZSAVE) yet.

To determine if a given program name is unused, programmers ZLOAD it. But if it does exist, again, they lose their unfiled program. One solution would be a $D (and $NEXT etc.) of program names. Even better, we could ZS *PROG which says to file this partition's program as PROG but only if it doesn't already exist.

Job Control

Signing on can be dangerous since the same input is used to start and to interrupt a program. They should be different. Also, it should be possible to interrogate a device without disturbing its processing.

Being able to execute into another processing partition would allow us to release a device after a program is started (for example, if it takes longer than expected), move it to another device, produce a clear symbol table listing from it, stop it, suspend it, or sign off unattended terminals gracefully by answering their reads. Simply executing OPEN (fake device number) CLOSE (same device number) after we opened it ourselves would allow us to suspend and resume any program without any additional programming, and control it from a single utility program.

ZGO should be allowed after a program is interrupted by an error or by entry of Control C or BREAK.

If a background job (one with no device - created by (Z)JOB or CLOSE $I) bombs on an error without $ZE being set, it opens its principle device, but then halts. If it was started by a device in programming mode, it should return to that device (programmer) in programming mode.

The Caretaker can stop due to being the fatal CPU or Disk partition after a crash and (auto) re-start (and at night it is often the only active job running), by getting a <DKHER> disk hardware error, or by being accidentally Restored. The Operating System should (Z)JOB a program name Viewed into the System Table by a utility program (or just use program %) at midnight (which can HALT if a prior copy is still running).

Summary

In summary, we feel that MUMPS can be made more effective by providing features that facilitate procedures common in the development and maintenance of applications programs.

References

1. *American National Standard Mumps Language Standard.* (ANSI X11.1-1977) American National Standards Institute, New York, 1977.

2. MUMPS Development Committee, *American National Standard Information Systems Programming Language MUMPS,* BRS/MDC X11.1, 1983.

3. Volkstorf, C.S., Mumps 1983 Needs Second Look, *ComputerWorld* (to appear March, 1984).

■