How to Handle the DO Command in the Age of $TEST

by Frederick L. Hiltz, Stage Manager

Question: Why doesn't the DO command save and restore the caller's value of $TEST, as do extrinsic functions and the argumentless DO?

Answer: It's a matter of timing. The DO command came first. When M and its contemporaries (BASIC, LISP, FOCAL, and others) were invented, the subroutine concept, borrowed from assembler languages of the day, was simply to avoid repeated code. Variables were not scoped, that is, the calling and the called code "saw" the same set of variables. Likewise, status flags were universally available; MUMPS subroutines often returned their results in $TEST, and sometimes in the naked indicator!

This paradigm permitted compact, speedy interpreters and small but unintelligible application routines. MUMPS programmers of the 1970s performed marvelous feats with 8-Kbyte interpreters, 1-Kbyte routines, and 1-Kbyte for local variables.

By the early 1980s, language designers recognized the value of closed subroutines and functions. Segments of code could be reusable and robust if they had no effect on the program but the intended effect. BASIC's GOSUB...RETURN and MUMPS' DO...QUIT acquired actual and formal arguments (FORTRAN had them a decade earlier). The NEW command limited the scope of local variables to the subroutine that used them. Extrinsic functions returned a value, eliminating that use of $TEST.

Most important, programs composed of closed subroutines and functions can be intelligible. Why aren't they? That, as the professor said, is beyond the scope of this discussion, but a worthy topic in its own right.

The MUMPS Development Committee (MDC), designers of our language, faced a hard choice. Should the members change the DO command to save and restore $TEST, breaking tens of thousands of MUMPS routines, or should they apply the new paradigm only to new language features, making them inconsistent with DO? As we know, they chose the latter, and to this day you can start a good argumeNt at any MTA Annual Meeting about it.

We can't have it both ways, but the MDC is considering several proposals to improve the situation. They fall into three categories:

- Substitutes for $TEST—A system variable that works more or less like $TEST, but is saved and restored by all forms of DO and extrinsic functions, would be used in new code.
- NEW $TEST—Let the NEW command apply to $TEST as it does to local variables. Subroutines would use it before changing $TEST. Calling routines could not rely on it unless the programmer examines the subroutine.
- Scoping for $TEST—New syntax brackets the range of code affected by an IF statement. One example adopts the IF...ELSE...ENDIF construction found in other languages. Nesting is permitted: IF saves $TEST before changing it. ELSE is optional. ENDIF restores $TEST saved by the matching IF.

None of these is perfect, and the best way to handle the effect on $TEST of OPEN, READ, and LOCK is still not settled.

What do you think? Send your reasons to the Stage Manager c/o the managing editor at *M Computing*.

Frederick L. Hiltz, Ph.D., develops medical information system software at Brigham and Women's Hospital, Boston, Massachusetts.