Error Processing

by Roger Partridge

A large segment of the MUMPS community has expressed a desire for a standardized mechanism for dealing with errors. Over the years there have been several proposals and much discussion on how best to do this. This article discusses the Type B, Error Processing proposal and the Subcommittee Type A, Naming of Existing Errors proposal. Because all programmers must deal with errors, but few are expected to use all features of error processing directly, the discussion focuses more on the issues and implications surrounding the proposal and less on coding details. For more formal and comprehensive information, refer to the proposal itself, obtainable from the MDC.

Introduction

ANS MUMPS defines a number of cases where certain language constructs are either "reserved," "unspecified," or "erroneous." ANS 90 MUMPS does not define what happens when a MUMPS program breaches any of these boundaries. Because a large part of a MUMPS programmer's task is to produce routines that execute properly, they need and expect some kind of notice when something is wrong. Up to the present, MUMPS implementors have filled this gap with "Z" extensions that report and control the circumstances surrounding an error.

In this article, the term "error processing" refers to the actions that the proposals have the MUMPS language perform; the term "error handling" refers to the MUMPS code supplied by the programmer for invocation during error processing.

Why Standardize Error Processing?

Although many of us have written perfect programs on the first try, we've all encountered errors because of inadequate or incorrect specifications, mistakes in code written by someone else, cruel and unusual treatment by a user or operator, or even our own occasional slip. Errors in programs are a universal problem. One of the strengths of MUMPS is its pioneering place in the world of open systems. In order to get useful work done in MUMPS, the requirements for constructs beyond the standard are minimal and confined to relatively few areas, namely I/O control (except for database operations), error processing/debugging, and interfaces to the environment outside of MUMPS.

The MDC is actively working in all these areas. It has approved an MDC Type A specification for controlmnemonics and is considering related proposals currently to address the I/O issue. It has approved as MDC Type A the external call and the Structured System Variable (SSVN) specifications, which provide powerful tools for resolving the external interface issues. It is considering the error processing document for elevation to Subcommittee Type A and the naming of errors document for elevation to MDC Type A.

MUMPS users and implementors all have dealt with error processing. Thus we have the experience to cooperate in developing a sound standard error processing mechanism.

Here are the main reasons I've heard for not standardizing error processing, and some possible responses:

Objection: "We've done quite nicely without it so far."
Response: Many users apparently disagree.
Objection: "We have too much invested in existing nonstandard technology."
Response: Users get the benefit of more portability for adapting to the standards, implementors get the same benefit (but may not see it as so rewarding).
Objection: "We can only do so much, and other enhancements are more critical to MUMPS."
Response: This one is clearly a matter of taste, but an overwhelming majority of the MDC voted to make error processing a priority for the next (1993) standard.

What to Consider in the Standard

An initial goal for the error processing proposal was that it be able to emulate all implementation-specific error processing extensions presented to the MDC. The consensus of the task group that has hammered out the proposal is that it comes close. The only exception we are aware of is that DataTree's DTM-PC and DT-MAX have a mechanism for handling asynchronous errors that permits continuation after handling an error. The current proposals do not provide for continuability, an issue that is discussed in a later section.

Another goal was to minimize the cultural content of error reporting. The MDC is currently investing considerable effort to consider extensions and modifications to MUMPS that permit or assist in making applications easier to "internationalize" or transplant to other parts of the world. In the meantime, we wish to avoid adding anything to the standard that inhibits internationalization (now known to most of the typing-impaired as i18n).

What to Do with Errors

MUMPS programs may take several approaches to dealing with an error.

The most brutal approach is to simply terminate the process. This has the virtue of preventing an errant program (and perhaps the person using it) from causing any further damage.

The first approach that comes to the mind of a programmer is to enter direct mode in order to start debugging. Traditionally the MUMPS standard has not addressed the debugging environment (usually referred to as direct or programmer's mode, or the debugging or MUMPS shell). MUMPS implementations typically provide a shell invoked with the BREAK command or other means. Because in practice almost all language constructs (standard or otherwise) are available in both the normal run-time environment and the debugging shell, it has been convenient for the standard to ignore the details of direct mode.

One concern expressed about the proposal is that some people associate error reporting with the debugging shell and therefore consider it inappropriate for inclusion in the standard. However, a majority in the MDC feels that error reporting should be available to programs as well as programmers and should therefore be standardized. Tradition indicates that features in the standard will be available in direct mode.

Most product managers favor the approach of saving as much context information as possible and then terminating or restarting at a known point. This approach permits the separation in time and place of the error incident and any subsequent debugging attempt.

Eventually, some ambitious user or programmer wants to use the approach of automating the recovery from certain errors. The task group concluded that MUMPS users require the flexibility to invoke some program (maybe as short as a single BREAK or HALT instruction) that handles the error according to goals that vary with the circumstances.

How to Classify Errors

There are many types of error classification to consider. For example, most languages differentiate between compiler errors and run-time errors. (This seems unattractive in a language with the late-binding features of MUMPS that allow arguments and even code to be determined at run-time.) Other popular distinctions are anticipated versus unanticipated, and recoverable versus fatal. Given the discretion that MUMPS typically grants programmers and implementors, the proposals leave both of these classifications to the MUMPS programmer.

Two related types of classification that the task group discussed are continuable versus noncontinuable and restartable versus nonrestartable.

Continuable means that after the error has occurred and error processing has been invoked, the process may be able to resume execution at the next command after the one completed (perhaps unsuccessfully) before error processing. DataTree's DTM-PC and DT-MAX offer this capability for certain types of errors that are handled asynchronously. Because many of the examples used to argue for continuability seemed to be errors that are asynchronous to the command execution, some task group members proposed that such cases should be treated as events rather than errors. (The MDC has a task group working on event processing.)

A majority of the task group decided that at present it would be unwise either to define in the standard—for every error—whether error handling provides the ability to continue, or to leave that choice entirely to the implementors. In effect, the proposal will require DataTree to introduce an extension for specifically requesting a transfer to the point of continuation.

Restartable means that after the error has occurred and error processing has been invoked, the process may be able to resume execution at the command that caused the error. While the proposal does not provide for restart as a general case, it does permit the error handling program to transfer control at the stack level where the error occurred. This means that if the failing command is preceded on a line only by actions that can safely be repeated (in the simplest case, no actions), the error handler can use a GOTO to cause a restart.

Finally, the proposals implicitly address a classification relating to what MDC members call backwards incompatibility. Where the standard defines errors, the Naming Existing Errors proposal assigns specific error codes. These error codes imply that their associated errors will not be removed from the standard without very serious consideration and ample warning. Where the standard defines reserved or unspecified constructs, the proposals do not assign error codes. While the task group expects that implementors will assign error codes to some or all of these currently unspecified or reserved constructs, the absence of a standard error implies that future standards may enhance the language by using reserved elements or by specifying previously unspecified behavior.

Interestingly enough, this means that while the proposals describe what to do with all errors, including implementor-defined errors, they do not define very much in the way of syntactic errors. This has the additional benefit (at least we hope it is a benefit) of allowing implementors to advance the language by making extensions to MUMPS without having to notify or get permission of the MDC.

How Does Transfer of Control Work When an Error Occurs?

The Error Processing proposal specifies that the error transfer happens at the same stack level as the one on which the error is detected. This avoids requiring a new stack level and avoids the problems of managing the case in which the original error is that of exceeding the maximum available stack level. It also preserves the current stack level for examination, debugging, and possible retry. With the chosen approach, a minimalist conversion of existing error processing that automatically changes the stack depth must emulate the stack modification. The later section describing $ESTACK has a simple example of such an emulation.

The error vector is stored in $ETRAP and is in the form of MUMPS code that is to be executed when an error is detected. Error processing inserts this code in the execution stream followed by a QUIT: $QUIT ""QUIT. The purpose of the added code is to QUIT with a null argument if the current stack frame is executing an extrinsic (a condition that causes $QUIT to be TRUE), and otherwise, to QUIT with no argument. This permits easy implementation of the BREAK and HALT cases, as well as emulation of existing error processing that uses a code execution approach. With the proposed approach, a simple conversion method for existing error processing that uses entryref style specifications for vectors is to insert GOTO or DO commands in front of the entryref before assigning them to $ETRAP.

The proposal specifies that an error arising during execution of an error handler is recorded as if it occurred at the stack level above the original error. This is intended to limit the chances of the second error overlaying the information for the first error. After recording the error, MUMPS removes a stack level and attempts error processing at the next lower level. While this does not prevent all possible error-related recursion or looping, it should deal gracefully with those cases of faulty error handling that the task group anticipated as most common. Because they are invoked under abnormal conditions, error handlers need to be carefully constructed and thoroughly tested.

The proposal specifies that error handlers are nested explicitly by issuing a NEW command with $ETRAP as the argument. This permits complete replacement of the current $ETRAP value, not only for the current level but also back as far as the level at which $ETRAP was last NEWed. With the proposed approach, a simple conversion method for existing error processing that implicitly stacks the error vector with every DO, XECUTE or extrinsic is to always NEW $ETRAP before SETting it.

In summary, the proposal chooses a specification for $ETRAP that provides the most flexibility to emulate all the existing error processing approaches brought to the attention of the task group.

What Information Should Be Available for Analyzing Errors?

This area appears to have more content and perhaps generates more controversy. As you review the features described in this section, remember that many of them are actually required in some form to properly specify the mechanics of the operation of $ETRAP, and also that the task group was attempting to standardize access to what the majority felt were the commonly desired elements of information.

Errors are encoded in comma-delimited strings in $ECODE. All standard errors are numbers registered by the MDC and prefaced with the letter "M." The reason for using numbers is to minimize the cultural content in the standard. The prefix "Z" is assigned to implementors and the prefix "U" to users. $ECODE is equal to the null string when error processing is not in progress. When $ECODE is not equal to the null string, it should always start with a comma (,) so that the primary error is in the second comma-delimited piece, and so on.

$ECODE can automatically change to a list of one or more codes when MUMPS detects an error, and can also be SET by a program. SET $ECODE="" cancels the error processing "state," specifically the handling of errors while error processing is active. SET $ECODE="" also cancels the ability to access information about stack levels that have been removed during error processing. However, note that after SET $ECODE="", normal execution will complete the code from $ETRAP and the QUTTs that error processing inserted, unless the programmer takes evasive action with a GOTO. SET $ECODE=var, where "var" is not null, triggers error processing. Note that var should be of a form such as ",Mnl,Zn2,Un3," (where nl, n2 and n3 are numbers, and only one comma-delimited piece is needed); otherwise, the SET will cause, rather than specify, the recorded error.

The proposals do not provide for standardized error texts. (Remember, we're trying to be culturally neutral about things the user might see.) However, the task group expects that implementors will provide a mechanism for translating the codes to more user-friendly messages. Once the MDC gets MUMPS internationalized, perhaps it will specify a way to get the error text in the language of your choice.

The $STACK intrinsic special variable returns the current level of nesting of DOs, XECUTEs and extrinsics. The levels counted by $STACK correspond to the PROCESS-STACK model used in the standard as a way of describing transfers of control and their effects on variable management.

The $ESTACK intrinsic special variable counts stack levels as $STACK. However, when $ESTACK is an argument to a NEW command, the effect is to save its current value and give it a value of zero. $ESTACK's anticipated most common use is controlling which stack level to stop at when emulating existing error processing that automatically removes stack frames. For example:

```
SET $ETRAP="IF '$ESTACK DO ^%ET"
```

$ESTACK is somewhat less essential than the rest of the proposal, in that a local variable could replace it in its anticipated most common use.

The $STACK intrinsic function provides access to information that describes the actions that brought the program to the last error, if error processing is active, and otherwise brought the program to the current command. These actions include DO, XECUTE and extrinsic nesting, and also any errors. $STACK( ) can have one or two arguments. The first argument is evaluated as an integer expression specifying a stack level. If $STACK() has a single argument of -1, it returns the highest level for which the two-argument form will currently return a non-null result. When error processing is not active, $STACK(-1)=$STACK. When error processing is active, $STACK(-1) may be greater than SSTACK, indicating that error-trace information is available for stack levels removed in the course of error processing. If $STACK() has a single argument of 0, it returns an implementation-specific value that the task group intends should show something about how MUMPS was invoked. If $STACK() has a single, positive, non-zero argument, it returns either the command word (DO or XECUTE) that invoked the level specified by the argument, or "$$", if the level was invoked by an extrinsic.

The second argument to $STACK( ) must evaluate to "PLACE" to retrieve the location of the current command at the specified level (as an approximate character offset) in its line (in entryref form) or in its XECUTE string (identified by an "@"). The second argument to $STACK () must evaluate to "MCODE" to retrieve the line, if available, or XECUTE string containing the current command at the specified level. The second argument to $STACK() must evaluate to "ECODE" to retrieve any errors detected at the specified level.

The proposals do not add any features for examining variable context information (including intrinsic special variables or SSVNs). The task group anticipates that many needs for this information can be handled by standard means such as $ORDER() or by implementation-specific extensions such as ZWRITE. The proposals do not provide a standard mechanism, other than removing stack levels, for a program to access variable states that have been NEWed. While as a programmer I can understand the usefulness of such a mechanism in certain debugging situations, I'm not yet convinced that the standard should recognize it as a legitimate programming practice.

Summary

The current error processing proposals would provide standard:

- Error identification format (in $ECODE).
- Errors for conditions that are declared erroneous by the standard.
- Program-defined transfer of control when an error is encountered (to code previously established by SET $ETRAP).
- Explicit stacking of error handlers (NEW $ETRAP).
- Information about how the program arrived at its current point or at the last error (in $STACK).

- Declaration or simulation of errors (SET $ECODE).Counting of stack frames from a parucular point (NEW$ESTACK).
- Means of determining if the current frame was invoked as an extrinsic ($QUIT).

Conclusion

These proposals represent a point in a long (about ten years), difficult, and expensive exploration of this problem by much of the MDC, all MUMPS implementors, and many MUMPS power-users.

The people involved in this process are too numerous to properly credit, but I'd like to mention some of the more prominent. If the proposals have elegance, much credit should go to Harlan Stenn, although he might wince at what we did to it after he became occupied with other things. If the proposals are clear, much credit should go to Thomas Salander who has edited the document in many lands and at unspeakable times. If the proposals are usable, much credit should go to Mark Berryman, who has patiently chaired endless meetings and explained problems and solutions to all sides.

Most implementors seem to feel that simply removing the Zs from the elements of their particular error processing facilities would achieve the optimum result. Some users feel the proposals are too complex, while others feel that they are incomplete. It is not yet clear that these proposals have the votes to become part of the standard. Consider whether you need their features, whether you like the proposals, how you might improve them (remember that they need the votes to pass), and mail, phone, e-mail or deliver in person your views to the MDC, or to one or more of its members.

Roger Partridge is vice president of operations for Greystone Technology Corporation, 100 Unicorn Park Drive, Woburn, MA 01801 (617-937-9000). He has had twenty years' experience with MUMPS technology. An active member of the MDC, Partridge has most recently served as the group's vice chairman (1991).