TO: MSPM Distribution
FROM: Karolyn Martin
SUBJ: Error Handling, BY.11.00
DATE: May 26, 1967

Sections BY.11.00 - BY.11.04 are being re-issued to reflect the following changes in error handling:

1. The name of the condition used to signal an error no longer contains a serial number; only one such signal is allowed per procedure (BY.11.00).

2. The error line is formatted in printerr (BY.11.04), not in seterr (BY.11.01).

3. There are two procedures instead of one to obtain error information in another procedure (BY.11.02).

4. Errors which the user has requested be deleted, but were not deleted because of an option setting, are marked differently (BY.11.03).

5. The users error file resides in the process directory of each process in the user process group.

6. An additional item of error information has been added: bit information in which such information as machine conditions may be recorded.

Published: 05/26/67
(Supersedes: BY.11.00, 10/07/66)

Identification

Overview of Error Handling
D. R. Widrig, K. J. Martin

Purpose

This paper discusses some of the philosophy and techniques involved in the notion of errors and error handling. Subsequent sections discuss specific implementation details for the Multics standard error handling mechanisms and procedures.

Discussion

Attendant to any computation center (and more prevalent in larger systems) is the notion of error. In the context of a computation system, one could loosely identify an error as the name given an event whose occurrence is undesirable. Note that this does not imply an unexpected event, it only implies a deleterious deviation from a planned event or chain of events.

In order to constrain the field somewhat, we shall limit ourselves to an investigation which includes only errors which are Multics-user caused, or are a by-product of some system malfunction. Specifically excluded are lost tapes, fire in the computer room, poor quality of a print ribbon, logical "bugs" in a user's program, etc.

While it is readily agreed that one cannot dispute the exact error in a given situation, (e.g., segment not found), it seems clear that there can be several interpretations as to the import, implication, or severity of the error. Moreover, such interpretations can also be altered according to the context of the imbedded error. To illustrate, suppose one wishes to eliminate segment A from the current working directory. It might seem reasonable for a delete command not to complain if A is not found. After all, the segment is no longer present after the command is completed and that is the objective of the delete command. On the other hand, the user may be vitally interested in knowing that the segment was not there in the first place. As another example, suppose a user wishes to input a segment X. Using the context editor, it is important

to know whether X already exists. If so, it is assumed by the editor that the user wishes to edit X. If, however, X were not found, then the editor presumes the user wishes to create a new segment. Thus, the editor is quite concerned over the existence of segment X.

A quite different concern may be found in the case of a compiler. In this case, the compiler may wish to delete Y so that it can replace it with a newer version from the current compilation. In this case, the compiler is unconcerned about the existence of previous versions of Y, it simply wants them all to vanish so Y can be replaced with a newer version.

In light of the above discussions, it seems clear that such "emotion" words as "fatal error", "warning", etc. must be subject to strict definition; otherwise, many conflicts of personal taste will arise.

We shall define a "fatal" error as one which precludes the continuing of a user's process on a normal basis. Note that this does not mean that being paged out of core is a fatal error, or that a compilation failed, or a segment was not available, etc. What is implied in the notion of a fatal error is the destruction of a user's file directory, inability to return to a standard listener, etc. It seems clear that the correct interpretation of a fatal error, from the user's viewpoint, is to abandon his current efforts and seek aid from competent personnel.

Another less disastrous but still "critical" error includes the occurrence of difficulties for which no remedial procedure is specified. That is, an anomaly is detected (in a subroutine, for instance) for which there are no prescribed "fix-ups" or error returns. One might envision this type of error occurring when a square root routine detects a negative argument. If no defaults are set and the caller specified no error return, the subroutine is faced with a problem since it cannot simply return with some arbitrary number. That is, from the subroutine's point of view, the problem is unsolvable.

It should be noted that many of the critical errors can readily be supplied with remedial tools. For instance, one could specify that the square root subroutine operates on the absolute value of the input arguments. Of course, such remedial tools must be clearly stated in program documentation and/or be alterable by the average user.

A third type of error is the so-called "failure" error. In this category, one raises the error condition if unable to perform the requested task or function. Although this definition allows a great number of possibilities, it does admit to some specific examples. For instance, at one level, one might declare a failure condition upon failing to find a segment, upon failure in accessing a segment, etc. At a much higher level, a command could signal a failure to perform its task. Such a mechanism will be invaluable in chains of commands where one could specify that the failure of command Y should terminate the chain. For example, if a compiler could not successfully compile a given source program, it might not be reasonable to proceed with the loading, etc.

Note that not all deviations from a standard situation should be considered to be errors. In some cases a number of possible situations may occur in a procedure, each of which requires a different action on the part of that procedure's caller. None of the situations precludes appropriate action by the caller, however. In this case, it may be expedient to include a return argument in the calling sequence with which status information can be returned to the caller, rather than handling low-probability situations as errors.

It seems clear that errors can occur in almost any module, procedure, or process throughout the Multics system. To avoid confusion, it is also apparent that a standard method of announcing errors is mandatory. Moreover, it is not unreasonable to suppose that the user may wish to create or define his own errors. For example, a user who has programmed a matrix inversion routine may wish to signal certain kinds of errors such as a zero determinant, loss of precision, etc. It follows that any error signalling mechanism adopted by the system should also be available for general use by the average user.

Another requirement for error handling is immediately evident; each different error requires a unique error code. For convenience, a simple numerical scheme will be used. For example, certain file system modules may establish that error 1 means a file cannot be found, error 2 implies that the file is already initiated, etc. It will be the responsibility of each module to maintain and manage their error codes, thus insuring uniqueness within a given module. A method is provided to record the code and other information about an error before announcing the error. Recording of this information is discussed on the following page.

Conditions in Multics are fault-like occurrences which necessitate action outside the normal flow of control. A procedure may state that upon the arising of one of these occurrences in a called procedure, it wishes to gain control and execute a block of code, that is, it is preparing for a possible situation. Either of the following EPL statements express this:

```
on condition (X) begin; or
call condition ("X", handler_procedure);
```

When a called procedure wishes to cause one of these occurrences to happen, it executes either of:

```
signal condition (X); or
call signal ("X");
```

Any procedure which has previously prepared for the signalling of this particular condition will "catch" the condition. If no procedure wants to take particular action when the occurrence arises, a default handler is provided by the system. See BD.9.04 for a complete discussion of condition handling in Multics.

To provide a standard method of announcing errors, the detection of the error is announced by an EPL statement of the form:

```
signal condition (subroutine_name_err); or
```

a call to the supervisor procedure:

```
call signal ("subroutine_name_err");
```

Thus, an error in the matrix inversion routine might be signalled by:

```
signal condition (matrixinvert_err);
```

One slight inconvenience may arise in using the above method. Since the signal statement in PL/I requires a standard identifier as the object of the signal, if the subroutine's name (concatenated with "_err") is longer than 31 characters, the user will have to call "signal" directly.

Because of the expense of enabling for conditions and signalling them, it is desirable to restrict the number of error conditions signalled by any one procedure. In general, procedures are restricted to one signal apiece for error purposes. The condition handler uses the

error code and other information to further distinguish between errors (see below and BY.11.02 for an explanation of error information).

The Shell will effectively operate with all error conditions enabled. (Since there are a great many possible error conditions, not all of which may be anticipated by a programmer, the signal processing subroutine signals the procedure unclaimed_signal which is part of the Shell in the case of detecting "unclaimed" signals.) Thus, if the user does not see fit to enable a particular error condition, any unplanned errors will automatically re-invoke the Shell. After the invocation, the user will undoubtedly wish to take corrective action. To achieve this goal, no system housekeeping will be done on the detection of the error condition since the user may wish to examine the stack, various data bases, active segments, and so on.

In order to properly analyze errors, it is important to know three basic items: (1) nature of the error, (2) location of error detection, and (3) caller of program detecting the error. In addition, it will frequently be useful to include extra or explanatory parameters which can be used by the error handler to properly interpret the remedial action to be taken. One explanatory item, the "error description", is mandatory. The error description consists of a character string, generated or maintained by the program detecting the error, which gives a helpful description of the error. This item will be found to be of great utility when making an analysis of a particular error. A typical error description might be the string "segment not found". Bit information may prove helpful in analyzing some errors. Additional character information may be useful to record such items as the symbolic label where the error occurred.

It appears that the following list represents minimal information necessary to catalog an error.

1. Location of call (caller)
2. Location of error detection (callee)
3. Error code
4. Error description
5. Extra bit-string information
6. Extra character-string information

In order to convey the error information to interested spectators, it is necessary to leave the material in a standard vector (or arrangement) in a standard repository. To this end, it shall be understood that the segment "error_out" in the process directory will receive all error messages associated with a given process. The error information in the error segment will be arranged in the following manner:

1. Calendar_time
2. Call_loc (Includes both the segment name and the offset)
3. Error_loc (Includes both the segment name and the offset)
4. Error_code
5. Error_info
6. Extra_bit_info
7. Extra_char_info

The following sections (BY.11.01 - BY.11.04) more fully describe the error-handling procedures which are:

1. Seterr - a procedure which formats error information and appends a complete error description at the end of the user's error segment for later use.

2. Geterr - a procedure which returns to the user the character information from the last error description found in the error segment. Geterr_complete - a procedure which returns all of the last error description.

3. Delerr - a procedure which deletes the last complete error description from the user's error segment.

4. Printerr - a procedure which takes all error descriptions or one selected description from the user's error segment, formats, and prints it.

The following example illustrates a straightforward application of the Multics standard techniques for error handling. Note that the actual occurrence of the error is an absolute event; however, the interpretation of the error is relative to the calling program and the entire embedded system the user has developed. Sections BY.11.01 - BY.11.04 should be referred to for explanation of calls to the procedures listed above.

Simple example of error handling

```
quadratic:    proc(a,b,c) float;

     /* This procedure returns the larger of two real
     roots of quadratic equations.  Complex roots are
     considered to be errors. */

     dcl (a,b,c,d) float, dummy char(0);
     dcl square_root ext entry (float) float;
     dcl who_called ext entry ptr;
     dcl neg_root label init(neg_rt);

     /* set error return for negative arguments */
     on condition (square_root_err) go to neg_root;

error_loc;    d = square_root (b*b - 4.0*a*c);

     return ((-b+d)/(2.0*a));

     /* error, negative square root, erase square root
     comment, set complaint from this subroutine */

     dcl code char (3) var init ("001"), info char(23)
         var init ("complex roots, bad data"), null_char
         char(0) var init ("");

neg_rt:       call delerr;

     call seterr (error_loc,code,info,null_char,null_char);

     signal condition (quadratic_err);

     /* if signal returns, return standard answer */

     return (0.0);

end quadratic;
```