Flexible Exception Handling in Forth-83

John Roye
(AT&T Bell Laboratories)
358 West 5th Street
Howell, NJ 07731

Abstract

This paper presents a Forth-83 implementation of flexible exception handling. Flexibility is achieved by allowing an exception to pass a value on the data stack so that the exception handling code can determine what to do. This makes selective handling and upward propagation of exceptions easy, without reference to global variables. It also allows many exceptions to be handled by one exception handler. This Forth-83 implementation is designed to be transportable across systems since it relies on only one simple system dependent word (`RP@`).

Background

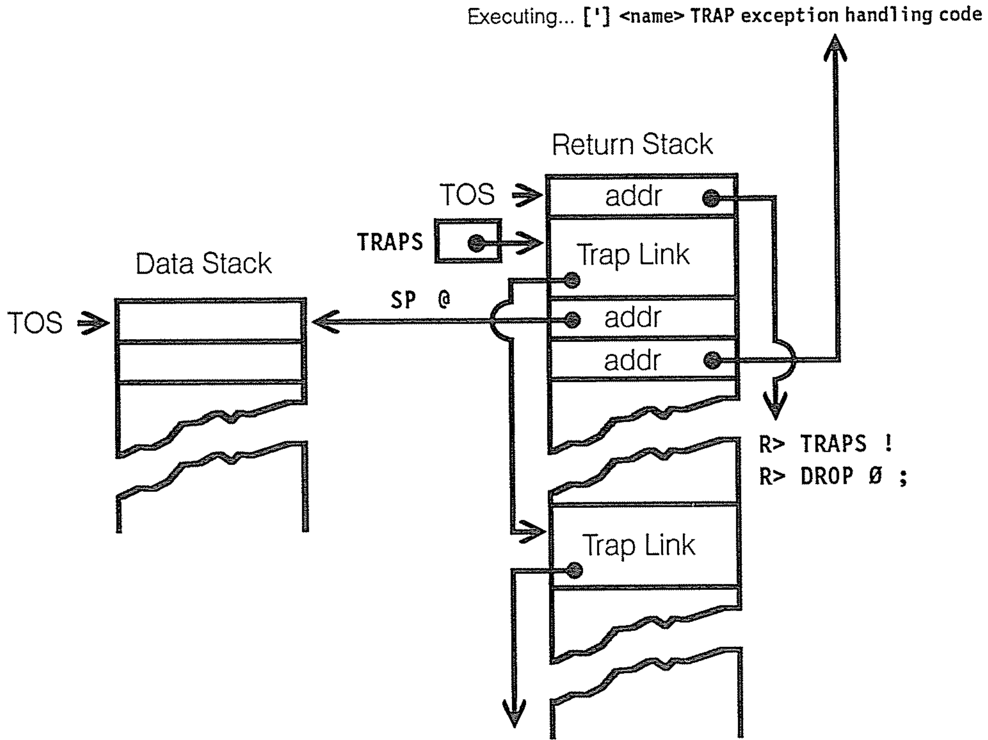
Many situations require the application code to recover from an exception rather than performing an abort. Sophisticated human interfaces are a prime example. Other examples are data sensitive cases (such as divide by zero) and perhaps expert system implementations. In any of these situations the coding can get messy in a hurry due to the existance of "nag flags" that must be tested throughout the code to determine whether a particular exception has occurred. The concept presented here reduces the need for such flags and tests to a minimum.

There are a number of articles in the Forth literature, published since 1981, that deal with exception handling in Forth ([BOU81], [SCH83], [SCH84], [COL83], [PAU85], [GUY86]). The most recent article, by Clifton Guy and Terry Rayburn [GUY86], is conceptually similar to the one presented in this paper. At the end of this paper their innovative solution will serve as a standard of comparision for the method presented here.

This implementation contains elements derived from the excellent work by Klaus Schleisiek [SCH83] based on dynamically linked stack frames anchored by a variable.

Implementation

Any high-level Forth word can be thought of as the root node of a tree of words. From within a tree of words the word `EXCEPTION` causes execution to return to the exception handling code following the root word `TRAP`. The word `TRAP` defined in screen 1 makes the code following `TRAP` the return point for exception handling as well as normal returns by the Forth inner interpreter. `TRAP` is used in the form ... `['] <name> TRAP exception handling code...`



In figure 1 the result of executing ... `['] <name> TRAP exception handling code...` is shown. Four items make up the software trap. The top item is the address of code that removes the trap frame, updates the variable **TRAPS** to point to the previous trap frame, and places a zero flag on the data stack. This top item is only executed in the case of a normal return (i.e. not caused by an exception) via the Forth inner interpreter. The second item is the trap-link. The trap-link links together a linked list of trap frames which are anchored by the variable **TRAPS**. **TRAPS** points to the currently active, i.e. most recent, trap frame (For a more in depth discussion see [SCH83]). The third item is a pointer to the top of the data stack at the time **TRAP** was executed. Finally, the fourth item is the return address to the exception handling code. This return address is the return point in all cases.

Implementation Discussion

The virtue of this approach is that if you do a **CLEAR-TRAPS** first you will always have a sensible response to the use of **EXCEPTION**, even when **TRAP** has not first been executed. A related benefit is that exception propagation to higher level exception handlers can be done securely. **TRAPS @** can be the condition allowing propagation of an exception upwards. A zero would indicate a program error (The exception number has propagated all the way up without being handled in any of the exception handlers). A minor problem with this approach is the use of the non-standard **RP@** which returns the address of the top of the return stack. One remaining point of possible controversy is that the word **EXCEPTION** resets the data stack to its state just prior to executing **TRAP**. Thus far I have found this useful. However, leaving this address on the data stack for the exception handler to use at its discretion may be a viable alternative.

One more small note. The code that resets the top of the data stack (lines 10 & 12 in screen 2) uses the phrase `SP@ R@ U<` as the terminating condition in the `WHILE` loop. `U<` assumes your data stack grows downward in memory. Should your stack grow upward you must write `SP@ R@ U< NOT`.

Comparision To Another Solution

For a discussion of other solutions to the general problem of exception handling in Forth see [GUY86]. Guy & Rayburn's efforts will not be duplicated here.

There are some definite conceptual similarities between the `EXCEPTION TRAP` concept and the `ESCAPE ALERT-EXCEPT-RESUME` constructs (These similarities are unique to these two approaches).

1. A specific exception is not bound to a specific exception handler.
2. Both methods allow nesting to an indefinite depth.
3. When an exception is performed both the data & return stacks are restored to the state of the higher level (i.e. at the exception handler level).

There are a couple of conceptual differences between `EXCEPTION TRAP` and `ESCAPE ALERT-EXCEPT-RESUME`.

1. By allowing `EXCEPTION` to return a value on the data stack it is possible to establish a many to one mapping of exceptions to exception handlers. This makes the selective handling and upward propagation of exceptions easy without reference to global data structures. To selectively handle or propagate exceptions with the `ESCAPE ALERT-EXCEPT-RESUME` construct is not possible without reference to global data structures.
2. In a technical as well as conceptual sense the `EXCEPTION TRAP` method does not violate the structured flow of execution. The Forth inner interpreter always returns to the same address. This is true whether the return is normal or via an exception. As a result there is no need to define a new control construct such as `ALERT- EXCEPT-RESUME` to hide the structure violation.

There are definite implementation differences. As Schleisiek [SCH84] points out "the return stack is the proper place to store information which has a limited lifetime corresponding to a certain execution level". `EXCEPTION TRAP` uses the return stack instead of a separate exception stack. The other major implementation difference is in how the normal versus the exceptional return is handled. By allowing a value to be returned in all cases on the data stack the exception handling code can determine what to do. The lack of this feature in the `ESCAPE ALERT-EXCEPT-RESUME` approach makes it necessary to build a new control structure `ALERT-EXCEPT-RESUME` which has the purpose of dividing the flow of execution into the normal clause `ALERT-EXCEPT` and the exception clause `EXCEPT-RESUME`. After comparing the `EXCEPTION TRAP` approach with the `ESCAPE ALERT-EXCEPT-RESUME` approach it would seem both display the same fundamental improvement over previous methods. Both methods hide the details of exception handling so that the application code reads in a more coherent way.

An Example

Screens 3, 4, & 5 are an example written in F83 that demonstrates the use of the words `EXCEPTION TRAP`. The F83 dependent words defined in terms of their stack effect are :

`CC` is a variable used to point to current control character table.

`DEL-IN` (n char — n-1)

erase previous char if n not zero.

CHAR (addr n char — addr n+1)
append char to input buffer.

PERFORM (addr-of-cfa —)
same as **@EXECUTE** but no test for zero.

NUMBER? (addr — d t=successful)

CASE: (n —)
executes the nth word at run time. At compile time requires the # of executable words on stack for the word being defined.

RP@ (— address-of-top-of-return-stack)

This example is not meant as anything other than a way of showing how the **EXCEPTION TRAP** constructs work. Note how easy it is redefine **EXPECT** to allow the user to escape at any time.

Summary

A new method has been described which is a general solution to the problem of exception handling in Forth. This method defines the new words **TRAP** & **EXCEPTION**. I believe that **EXCEPTION TRAP** is simpler and to a degree more general than the **ESCAPE ALERT-EXCEPT-RESUME** concept. It is certainly more transportable across Forth-83 systems since it relies on only one non-standard primitive word **RP@**. In any case I hope the reader will find the concepts presented here to be useful.

References

[BOU81] Boulton, David, "**UNRAVEL** and **ABORT:**" Improved Error Handling for FORTH.", *1981 FORML Conference Proceedings*, Vol. 1, pp. 161–165. San Jose, CA : FORTH Interest Group, 1982.

[SCH83] Schleisiek, Klaus, "ERROR TRAPPING: a Mechanism for Resuming Execution at a Higher Level.", *1983 FORML Conference Proceedings*, pp. 151–154. San Jose, CA : FORTH Interest Group, 1984.

[SCH84] Klaus Schleisiek, "Error Trapping and Local Variables", *1984 FORML Conference Proceedings*, San Jose, CA : FORTH Interest Group, 1985.

[COB83] Colburn, Don, "User Specified Error Recovery in FORTH.", *1983 FORML Conference Proceedings*, pp. 159–164. San Jose, CA : FORTH Interest Group, 1984.

[PAU85] Paul, Robert J., Friedland, Jay S., and Sagan, Jeremy E., "Run-Time Error Handling in FORTH Using **SETJMP** and **LNGJMP** for Execution Control or GOTO in FORTH." *Journal of FORTH Application and Research*, Vol. 3, No. 1, pp. 83–85. Rochester, NY : Institute For Applied FORTH Research, Inc., 1985.

[GUY86] Guy, Clifton, Rayburn, Terry, "Exception Handling In FORTH", *Journal of FORTH Application and Research*, Vol. 3, No. 4, Rochester, NY : Institute For Applied FORTH Research, Inc., 1986.

John Roye received a BS in Physiological Psychology in 1975 and a BS in Computer Science in 1985 both from Arizona State University. John recently authored a patent application for a very fast 2-D object processor for the GTX corporation. In 1988 he completed an M.S. in Computer Science at the National Technological University. His professional interests include graphics applications, image processing, and real time systems. John is currently a member of the technical staff at AT&T in Middletown, New Jersey.

```
( Exception handling by John Roye is in the public domain
  and may be reproduced with this notice. )
( Screen 1                         TRAPS, CLEAR-TRAPS, RP@, TRAP )

VARIABLE TRAPS
: CLEAR-TRAPS  ( --- )  Ø TRAPS ! ; CLEAR-TRAPS

CODE  RP@ ( --- addr )   \ F83 8Ø86 code for IBM PC
 BP PUSH NEXT END-CODE

: TRAP  ( cfa --- exception# )
  >R SP@ R> SWAP >R       \ Put SP@ in frame.
  TRAPS @ >R              \ Put trap-link in frame.
  RP@ TRAPS !             \ Make TRAPS point to this frame.
  EXECUTE  R> TRAPS !     \ Make TRAPS point to previous frame.
  R> DROP Ø ;             \ Drop SP@ & leave a Ø flag.

( Screen 2                                           EXCEPTION )

: EXCEPTION  ( [n items] exception# --- exception# )
( Insure TRAPS has been set by TRAP. )
  TRAPS @ ?DUP Ø= ABORT"  TRAP not set."
( Unravel return stack. )
  BEGIN R> DROP DUP RP@ = UNTIL  DROP
( Make TRAPS point to previous frame. )
  R> TRAPS !
( Prepare to unravel data stack. )
  R> SWAP >R >R
( Unravel data stack. )
  BEGIN SP@ R@ U< WHILE DROP REPEAT  R> DROP
( Leave exception# on data stack. )
  R> ;

( Screen 3               EXCEPTION TRAP example written if F83. )

1 CONSTANT #INVALID
2 CONSTANT ZERO/
3 CONSTANT ESC

: EXPECT/EXCEPT  ( addr len --- )
( Same as EXPECT except does an ESC EXCEPTION on an ascii 27. )
  DUP SPAN !  SWAP Ø
  BEGIN 2 PICK OVER - WHILE
    KEY DUP BL <
    IF DUP 27 = IF ESC EXCEPTION THEN
       DUP 2* CC @ + PERFORM
    ELSE DUP 127 = IF DEL-IN ELSE CHAR THEN
    THEN
  REPEAT   2DROP DROP ;
```

```
( Screen 4  EXCEPTION TRAP example continued. )

: VALID#  ( --- 16-bit# )
   PAD 1Ø BLANK  PAD 1+ DUP 1Ø
   EXPECT/EXCEPT SPAN @ -TRAILING PAD C! 1-
   NUMBER? NOT IF #INVALID EXCEPTION THEN DROP ;

: VALID#S  ( --- 16-bit# 16-bit# )
   CR ." 1st # : " VALID#
   CR ." 2nd # : " VALID# DUP Ø= IF ZERO/ EXCEPTION THEN ;

: DIVIDE   ( 16-bit# 16-bit# --- )
   CR 2DUP SWAP . ." / " . /MOD ." = " . ." rem " . CR ;

: INVALID# ( --- )
   CR ." This # not a valid #. Please re-enter." CR ;

( Screen 5                 EXCEPTION TRAP example continued. )

: DIV/Ø  ( --- )
  CR ." You attempted a divide by zero!" ;

: USER-DONE  ( --- )
   CR ." Good bye!! " R> R> 2DROP ;

4 CASE: EXCEPTION-HANDLER  ( exception# --- )
   DIVIDE INVALID# DIV/Ø USER-DONE ;

: EXAMPLE   ( --- )
   CLEAR-TRAPS CR ." This example divides two numbers." CR
   ." To quit at any time press <Esc> (i.e ascii 27)."  CR
   BEGIN ['] VALID#S TRAP EXCEPTION-HANDLER AGAIN ;
```