Exception Handling in FORTH

Clifton Guy and Terry Rayburn

Bremson Data Systems, Inc.
11691 W. 85th St.
Lenexa, KS 66214

Abstract

FORTH relies on the discipline of the programmer to provide the benefits of structured languages: readability, predictability and modularity. A difficult class of problem for structured software is the handling of exceptions to the control flow, including errors. Even in properly designed software, a low level module may detect a condition whose proper resolution resides on a higher level. We describe a general implementation for exception handling in FORTH that hides the inherent structure violation within a readable control structure which allows return from any nesting depth of FORTH words and control structures. Further, this structure is itself nestable to any arbitrary depth.

Exception Handling in Structured Programs

The goals of structured programming are to improve the clarity of presentation of the control flow and to standardize style. A limited number of control structures have been described as the primitives in whose terms all algorithms can be expressed. These primitives are the sequence, the `DO-WHILE` and the `IF-THEN-ELSE` structures. Various camps now support the inclusion of the `CASE` construct, the `BEGIN-UNTIL`, the `BREAK` and the `LEAVE` in the legitimate set [SHO83]. All consistently plead that the use of `GOTO` be minimized.

A common problem for structured programmers is the handling of control flow exceptions such as error conditions. In a well-structured program, there may be several layers of code between the user and the lowest level utility that may detect an error.

A typical approach is to pass flags back from the layer in which an error is detected. Each layer in the path looks at the flag and decides whether to exit or continue. A more sophisticated approach is to use a vectored abort, registering an error handler before the descent into the lower layers. Upon error detection, the code may execute an `ABORT` which will pass control to the registered handler. Although the former technique is technically structured, neither approach meets the goals of structured programming. We believe an exception handling construct should preserve the clarity of the control flow while providing a direct path from exception detection to the handler. Our goal is to present such a construct that can be added to a FORTH system without modifying the kernel.

Theory of the Solution

The solution to the problem of exception handling in a structured program ultimately involves unwinding the thread of execution as represented by the return stack. When an exception is raised, the mechanism could pop the return stack by some preset number of levels or unwind through the exits of the intervening words. In either case the return stack pointer is restored to the level of the exception handler. Therefore, it is possible to build a general denester by saving and restoring the return stack pointer. Since the parameter stack has been used by intervening words, its pointer should also be restored. This is essentially saving and restoring the context of the upper level.

The most useful version of the solution would be molded into the form of common control structures with which the programmer is already familiar. This new control structure would contain

a clause for handling the exception which we call the except clause. As execution begins its descent to lower layers, we wish to save a projected "future context" that will exist if an exception occurs. We must save the FORTH interpreter pointer modified to point to the except clause, which we call the exception address. Existing control structures such as **IF** work in just this way.

The future context may be saved on a stack to allow nesting of the structure. We call this stack the exceptions stack. With this implementation, exception handlers themselves may generate exceptions. Three items are saved on the stack in our implementation: the parameter stack pointer, the return stack pointer and the exception address. Other context items important to the application may also need to be saved. These would include a string stack pointer or user stack pointer, if implemented.

ALERT-EXCEPT-RESUME

The proposed control structure is:

```
ALERT  (ALERT clause)  EXCEPT  (EXCEPT clause)  RESUME
                .
                .
                .
            ESCAPE
```

The exception control structure looks like a familiar **IF-ELSE-THEN** or **BEGIN-WHILE-REPEAT**. Like the **BEGIN-WHILE-REPEAT**, all three elements are required for the structure. **ALERT** registers the **EXCEPT** clause as the exception handler. If an error is detected, **ESCAPE** raises the exception which passes control to the handler. If **ESCAPE** is not executed, the **ALERT** clause will complete normally and execution will continue following **RESUME**.

Implementation

On screens 1 and 2 we present an implementation of **ALERT-EXCEPT-RESUME** for polyFORTH II on a Digital Equipment Corp. PDP-11. The FORTH interpreter pointer is kept in a register named **I**. **S** is the parameter stack pointer and **R** is the return stack pointer. **NEXT** assembles the inner interpreter as two instructions. The standard word **END-CODE** is not used to terminate a **CODE** definition. Unlike the FIG model, polyFORTH does not insure that conditionals are paired.

```
0000 PREALERT
0002 (ALERT)
0004 0026 (exception address)
0006 alert clause
    .
    .
    .
0020 POP
0022 Branch to 0050
0026 except clause
    .
    .
    .
0050 execution continues
```

Figure 1. **ALERT-EXCEPT-RESUME** Memory Model

Run-time behavior

Figure 1 is a memory model of a compiled ALERT-EXCEPT-RESUME structure. The first word in the structure to execute is PREALERT (screen 1), which implements optional run-time checking. PREALERT verifies that there is enough room for the code word (ALERT) to push the context onto the exceptions stack. This word may be removed from the final application. We include it in our version because instances of exception stack overflow are difficult to trace. If overflow occurs, PREALERT aborts. This should occur only during testing of the application. The solution is to make the exceptions stack larger.

(ALERT) is the code word that registers the handler by saving the context on the exceptions stack. Figure 2 is a representation of the exceptions stack before (ALERT), after (ALERT) and after RESUME. S and R are pushed onto the exceptions stack. The interpreter pointer points to the word following (ALERT) when (ALERT) executes. The exception address has been previously compiled at that location. This address is fetched and pushed onto the exceptions stack. The interpreter pointer is incremented by two to skip over the address. At this point the ALERT clause is executed.

If an ESCAPE is executed, it must restore the future context saved by (ALERT). It pops the exception address into I and restores R and S. Upon execution of NEXT, control passes to the EXCEPT clause. If the ALERT clause completes without an ESCAPE, the word POP will pop the exceptions stack, and the next word branches around the EXCEPT clause.

Compile-time behavior

ALERT is an IMMEDIATE word that compiles PREALERT and (ALERT). (ALERT) does not call PREALERT since calling a colon definition from a code definition is obscure. If PREALERT called (ALERT), the return stack pointer would have changed and necessitated an adjustment before being pushed. A useful by-product of this implementation is the easy removal of the run-time checking. Finally, ALERT must compile a cell to receive the exception address that will be resolved by EXCEPT and leave the address on the parameter stack.

EXCEPT compiles POP and executes ELSE. ELSE resolves the exception address into the cell marked by ALERT. It compiles a branch and an empty cell and leaves the address of the cell to be resolved by RESUME. RESUME is a synonym for THEN which indicates the end of the control structure.

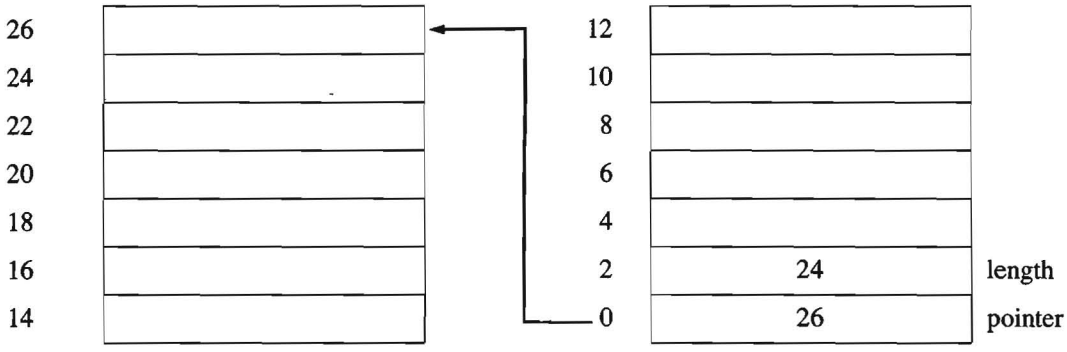
Application Notes

An example of ALERT-EXCEPT-RESUME is shown on screen 3. The physical I/O would take place where the comments "read block" and "write block" are found. A status word is checked for success or failure. In our hypothetical case there is no read or write, but we set the status word externally to demonstrate the behavior. The word COPIES is our application layer that reads a block, farbles the data where the comment "process" appears, and writes the block back out.

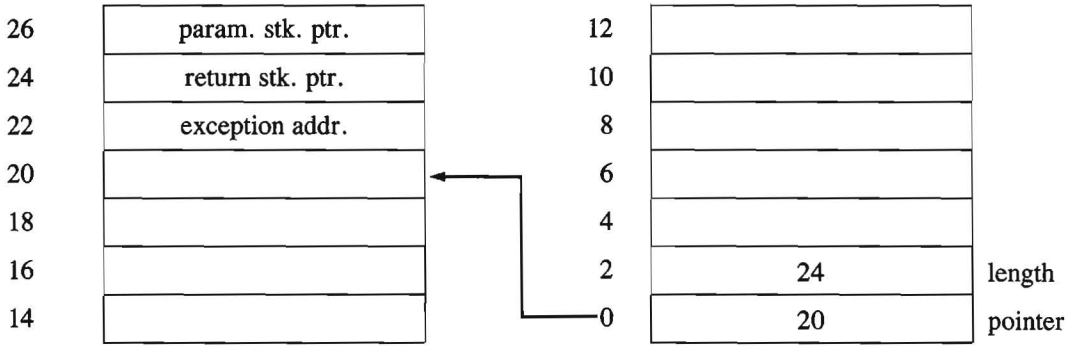
When EXAMPLE executes and both READ and WRITE are successful, then a success message is printed. If either READ or WRITE fails, then ESCAPE causes control to pass to the EXCEPT clause which prints an error message. Note that ESCAPE is two levels below EXCEPT and nested inside a DO-LOOP.

For experimenters: ESCAPEs could also appear at other levels, perhaps as a range check on the loop parameters in COPIES. As we have mentioned, ALERT-EXCEPT-RESUME blocks may be nested. In such a case an ESCAPE could appear within the EXCEPT clause of our example to pass control to a higher level handler.

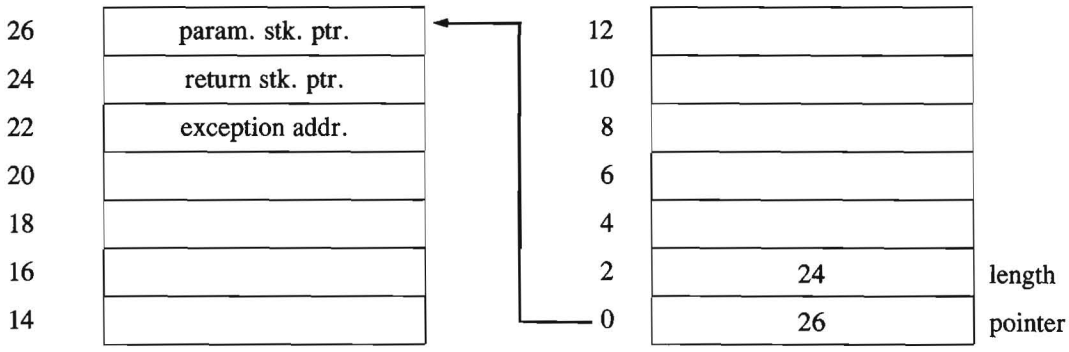
During incremental testing of our application, we often executed a word from the interpreter that used ESCAPE. If an error condition occurred, the program would crash ungraciously since the exceptions stack had not been set up. We found it helpful to redefine ESCAPE to give an operator message and QUIT, which returns control to the programmer with the parameter stack intact.



EXCEPTIONS STACK before (ALERT)



EXCEPTIONS STACK after (ALERT)



EXCEPTIONS STACK after ESCAPE or POP

Figure 2. Exceptions Stack Diagrams

We have referred to our process as exception handling and have given an example of the special case of handling errors. A colleague has suggested that the mechanism could be used to implement a forgiving user interface. Such an interface would allow the user to escape from the current mode by pressing a particular key, perhaps the ASCII "escape". An **ESCAPE** would return control to some predictable level no matter how convoluted a path the user had taken to his current predicament.

Comparison to Other Solutions

Other solutions to the general problem of exception handling have been described in the FORTH literature. Joosten [JOO82] and Nieuwenhuijzen [NIE82] describe a modification to **ABORT** that causes it to search the compiled code of each word in the calling sequence looking for an error recovery marker. When the marker is found, **ABORT** causes execution to resume at the following location. This marker is the compilation address of a synonym for **EXIT**. So, if the marker is reached during normal execution, the result is an exit. **QUIT** has been modified to contain an error handler of this type. If no intervening handler is found, the handler in **QUIT** will be found and executed.

This solution has several features to commend it. It is easy to use and highly readable. Putting an error handler into **QUIT** is a good idea. **ALERT-EXCEPT-RESUME** could make use of this idea to insure that a handler is always registered. One elegant characteristic of this solution is that it does not require explicit registration of error handlers. However, unless **ABORT** is very smart, the mechanism is subject to failure. One problem area is that data such as loop indices kept on the return stack are indistinguishable from return addresses. This can cause a search for an error recovery marker in an area of memory unrelated to the thread of execution. The second problem area is that data in colon definitions other than compilation addresses may be mistaken for the error recovery marker. Examples of such data are literals, strings, branch offsets and whatever in-line extensions the user may add. This last item makes it hard to provide a general solution by making **ABORT** smart. Joosten reports that these problems do not appear to matter in practice.

Schleisiek [SCH83] provides a defining word for writing named exception handlers which when executed will return control to a higher level. Registering a named handler has two effects. It enables the execution of the handler if the exception occurs and it marks the place where execution will resume afterwards. This solution is portable because it is implemented entirely in high level FORTH.

Having a named handler for a specific exception is an attractive idea. In our solution, a specific exception condition is not bound to a specific handler. If the programmer were to insert an **ALERT-EXCEPT-RESUME** block between the layers of an **ESCAPE** and an existing exception handling block, that new block does not handle the newly defined exceptions alone. Instead, the existing lower level exceptions will pass control to the new **EXCEPT** clause. These must be detected and sent on. Despite the benefits of tightly coupling the exception and its handler, we believe that the ultimate disadvantage of Schleisiek's syntax is that it places all of the responsibility for an understandable structure on the application programmer when this could be easily assumed by the underlying mechanism.

Schleisiek puts the exception information in linked return stack frames. We agree that "the return stack is the proper place to store information which has a limited lifetime corresponding to a certain execution level" [SCH84]. A better implementation of **ALERT-EXCEPT-RESUME** would use return stack frames instead of the exceptions stack. The separate data structure is unnecessary.

Colburn [COL83] describes an approach to error handling that vectors the behavior of **ABORT"** via a frame on the return stack. In a way similar to **ALERT-EXCEPT-RESUME**, his approach codes and registers the exception handler in-line. An **ABORT"** occuring within the scope of this definition will branch to the registered handler. Execution resumes following the exception handler, which is coincidently the code that caused the exception to occur in the first place. This provides an automatic retry mechanism.

While we restricted ourselves to a solution that could be layered onto an existing FORTH, Colburn modified **ABORT"** to his benefit. First, **ABORT"** raises the exception, performing the function of our **ESCAPE** without adding a new word. Second, **ABORT"** behaves as usual if no handler is registered. We like the idea of having a default error handling behavior and having it coded in-line. This allows the word detecting the error condition to be written and tested without any external scaffolding.

The problem with Colburn's approach is that it results in code that is unnecessarily difficult to read because it does not resemble any familiar structure. The inherent retry mechanism seems at first a benefit that would compensate for the loss of readability. However, the complexity that seems to be saved must be inserted in order to make the mechanism stop retrying. Placing an **ALERT-EXCEPT-RESUME** within a **BEGIN-UNTIL** block provides an explicit retry loop that is easier to understand.

A Final Word

We have described a general solution to the problem of exception handling in FORTH. It defines the words **ALERT**, **EXCEPT**, **RESUME** and **ESCAPE**. We have also compared our solution to several others which have been described in the literature. We like the simple syntax described by Joosten and Nieuwenhuijzen but are concerned by the frailties of the approach. There are also some attractive aspects of the solutions described by Schleisiek and Colburn but both suffer from a lack of readability. Since **ALERT-EXCEPT-RESUME** follows the form of other control structures, it can be easily combined with them to provide any general control flow and exception behavior. We regret that the implementation we present does not use the return stack frame model for storing exception information. However, modifying the code we have given to use the return stack should not be too difficult. We hope that the reader will find this structure a useful extention to his FORTH system.

References

[COL83] Don Colburn, "User Specified Error Recovery in FORTH," *1983 FORML Conference Proceedings.*

[JOO82] Rieks Joosten, "Techniques Working Group (report)," *1982 Rochester FORTH Conference Proceedings.*

[NIE82] Hans Nieuwenhuijzen, "The Importance of the Routine QUIT," *1982 Rochester FORTH Conference Proceedings.*

[SCH83] Klaus Schleisiek, "Error Trapping: A Mechanism for Resuming Execution at a Higher Level," *1983 FORML Conference Proceedings.*

[SCH84] Klaus Schleisiek, "Error Trapping and Local Variables," *1984 FORML Conference Proceedings.*

[SHO83] Martin L. Shooman, *"Software Engineering Design/Reliability/Management,"* McGraw-Hill, Inc., 1983.

Manuscript received October 1985.

Terry Rayburn received the BS in Physics from Texas A&I University in 1971 and the MSEE from the University of Missouri in 1983. He has used FORTH as a consultant in a variety of applications and is currently a Software Engineer with Bremson Data Systems in Lenexa, Kansas.

Clifton Guy received the BS in Computer Engineering from Iowa State University in 1984. A Systems Design Engineer at Bremson Data Systems, his interests include software engineering, data structures, and communications.

Glossary

(ALERT) is the execution-time code of **ALERT**. It pushes three values onto the exceptions stack. These are the parameter stack pointer, the return stack pointer and the "future" FORTH instruction pointer.

ALERT is an **IMMEDIATE** word which begins an exception handling control structure terminated by **RESUME**. Besides compiling **PREALERT** and **(ALERT)**, it has effects similar to **IF**. It leaves an empty cell in the dictionary to be filled in by **EXCEPT** which will contain the address of the **EXCEPT** clause.

ESCAPE restores the parameter and return stack pointers, and resets the interpreter pointer **(I)** to point to the **EXCEPT** clause.

EXCEPT is an **IMMEDIATE** word which compiles **POP** in order to pop the exceptions stack if the **ALERT** clause completes normally. It uses **ELSE** which fills in the empty cell left by **ALERT** and leaves an empty cell to be filled in by **RESUME**.

EXCEPTIONS is a data structure that contains a stack pointer at **EXCEPTIONS+0**, the stack depth at **EXCEPTIONS+2**, and a stack at **EXCEPTIONS+4**. This stack is used to hold information needed at the time **ESCAPE** is executed. The stack allows nesting of **ALERT-EXCEPT-RESUME** blocks up to 4 levels deep (3 words per level is 24 bytes).

POP pops three items off the exceptions stack.

PREALERT verifies there is enough room on the exceptions stack for **(ALERT)** to push its three values.

RESUME is an **IMMEDIATE** word which is a synonym for **THEN**. It fills in the empty cell left by **EXCEPT** and marks the end of the control structure.

```
SCREEN 1
( Exception handling by Clifton Guy and Terry Rayburn is in
  the public domain and may be reproduced with this notice)

VARIABLE EXCEPTIONS  24 DUP ,  ALLOT  HERE EXCEPTIONS !

: PREALERT
   EXCEPTIONS @  EXCEPTIONS  10 +  < IF
        EXCEPTIONS  DUP 2+ @ +  4 +  EXCEPTIONS !
        1 ABORT" Exceptions stack overflow"
   THEN ;

CODE (ALERT)
    0 EXCEPTIONS MOV    0 -) S MOV    0 -) R MOV
    0 -) I )+ MOV    EXCEPTIONS 0 MOV
NEXT


SCREEN 2
( ESCAPE  ALERT  EXCEPT  RESUME )

CODE ESCAPE
    0 EXCEPTIONS MOV    I 0 )+ MOV    R 0 )+ MOV
    S 0 )+ MOV    EXCEPTIONS 0 MOV
NEXT

: ALERT    COMPILE PREALERT COMPILE (ALERT) HERE 0 , ;
    IMMEDIATE
: POP    6 EXCEPTIONS +! ;
: EXCEPT   COMPILE POP  [COMPILE] ELSE ;  IMMEDIATE

: RESUME   [COMPILE] THEN ;  IMMEDIATE


SCREEN 3
( ALERT-EXCEPT-RESUME example )

VARIABLE RSTATUS    VARIABLE WSTATUS

: READ   ( read block )  RSTATUS @ NOT IF  ESCAPE  THEN ;
: WRITE  ( write block)  WSTATUS @ NOT IF  ESCAPE  THEN ;

: COPIES  ( n - )  0 DO  READ  ( process )  WRITE  LOOP ;

: example  ALERT 3 COPIES  CR ." Copy successful."
           EXCEPT  CR ." Error in COPY."
           RESUME  ." Done." ;

: EXAMPLE  2 0 DO    2 0 DO
             I RSTATUS !  J WSTATUS !  example
           LOOP  LOOP ;    EXAMPLE
```