A Generalized EXIT

Carol Pruitt
105 Salem Road
Rochester, New York 14622

Abstract

Like **EXIT**, **{EXIT}** allows a Forth word to abort its own execution upon discovery of an exceptional condition. Unlike **EXIT**, however, **{EXIT}** can be used within a **DO** loop. It can specify the point where execution is to resume. And its behavior is not affected by refactoring of the code that uses it.

Background: Some Problems

The standard Forth **EXIT** is quite a useful word: it allows a colon definition to respond to exceptional conditions by aborting its own execution and returning immediately to the word that called it. It is most commonly used upon detection of a condition which renders execution of the remainder of the definition either meaningless or dangerous.

EXIT has its limitations, however. Both the 1979 and 1983 standards explicitly state that **EXIT** may not be used within a **DO** loop. From a practical standpoint, this is because it gets its destination address from the top of the return stack; traditional Forth systems also use the return stack for storage of the **DO**-loop counters, thus blocking **EXIT**'s access to the appropriate return address. Even though some current Forth systems have a separate **DO**-loop stack for this purpose, many do not, and the standard cannot allow a practice which does not work on all systems. The traditional substitution for **EXIT**-within-a-**DO**-loop is a comparatively complex structure of flags and **IF**s [figure 1].

Two other limitations may become apparent when a nest of Forth words is re-factored [figure 2]. If a word which calls **EXIT** becomes in-line code, or if a large word is factored into smaller words, **EXIT** may no longer go to the right place. Again, flags and **IF**s traditionally replace the **EXIT**s. The implementation must change, even though the logic has not.

Various special-purpose alternatives to the flag-and-**IF** technique have been proposed. **[BROD84]** (Chapter 8, "Minimizing Control Structures: Using Structured Exits") suggests direct manipulation of the return stack as one possibility, and an **EXIT**-type word specialized for **DO** loops as another. Neither of these solutions, however, allows the code to be restructured. Explicit return-stack manipulation fails if the level of nesting changes, and a word designed to be used inside a **DO** loop will not work if, for example, a **BEGIN** loop is substituted.

A Solution

A more general technique is possible. Although this technique could replace **EXIT** altogether, the simplicity and frequent usefulness of the plain-vanilla word make it worthwhile to retain the old **EXIT**. Therefore, the new word has a new name, **{EXIT}**, pronounced "curly **EXIT**." Since all the problems enumerated above are due to **EXIT**'s not knowing where to go, it is also necessary to mark the level where execution is to return if the exit occurs. The markers are named

{ and **}** (note that each is a Forth word and must be set off by spaces). Since the markers must always be used together, they can be referred to collectively as "curly braces," or individually as "left curly" and "right curly."

The use of **{EXIT}** can be demonstrated by applying it to the preceeding examples. In Figure 3, **{EXIT}** works within a **DO** loop just the way **EXIT** "should" have. Not only do all three examples in Figure 4 use **{EXIT}** in exactly the same way; all three examples actually accomplish the same thing. (Note in the second example that the in-line code from **DDD** is enclosed in curly braces just as the name of **DDD** is in the first and third.)

Like **EXIT**, **{EXIT}** can be used any number of times within the same definition. But because **{EXIT}** can be used at any level within a nest of words, it can actually be used at more than one level within the same nest [figure 5]. If the entire nest is enclosed by a single pair of curly braces, all **{EXIT}**s within the nest will return to the same right curly. When there are multiple paired braces, an **{EXIT}** will advance, just as one might expect, to the right curly corresponding to the most-recently-executed left curly. And one **{EXIT}** can even advance to any of several different right curlies, depending on what part of the code calls the word that contains the **{EXIT}**.

Some Limitations

The many advantages of **{EXIT}** are accompanied by a few, relatively minor, cautions. The first three are implementation dependent.

```
\ what we'd really like to do, if only we could:
: AAA
     whatever01
     DO whatever02
        IF EXIT THEN     \ if we exit,
        whatever03
     LOOP
     whatever04 ;
: BBB
     whatever05
     AAA
     whatever06 ;        \    we'd like to come here

\ what we end up doing instead:
VARIABLE flag            \ we define a flag
: AAA
     whatever01
     FALSE flag !        \ we initialize the flag
     DO whatever02
        IF TRUE flag !   \ we set the flag
           LEAVE         \    & leave the DO loop
        THEN
        whatever03
     LOOP
     flag @              \ then we test the flag
     IF EXIT THEN        \    and if it's set,
     whatever04 ;
: BBB
     whatever05
     AAA
     whatever06 ;        \    we come here
```

Figure 1.

When curly braces are placed within a **DO** loop, the loop index cannot be retrieved by using **I** within the braces, since the marker left on the return stack by left curly [see IMPLEMENTATION NOTES, below] will block **I**'s access to the index. The solution is to call **I** from outside the braces, leaving the index on the parameter stack for use from within. Curly braces outside a **DO** loop will not interfere with the use of **I**.

```
\ the original factoring:
: DDD
     whatever07
     whatever08
     IF EXIT THEN  \ if we exit,
     whatever09
     whatever10 ;
: EEE
     whatever11
     DDD
     whatever12 ;  \    we come here
: FFF
     whatever13
     EEE
     whatever14 ;

\ one refactoring that doesn't work:
: EEE
     whatever11
     whatever07
     whatever08
     IF EXIT THEN  \ if we exit,
     whatever09
     whatever10
     whatever12 ;  \    we'd like to come here,
: FFF
     whatever13
     EEE
     whatever14 ;  \    but we come here instead

\ and another that doesn't:
: CCC
     whatever08
     IF EXIT THEN  \ if we exit,
     whatever09 ;
: DDD
     whatever07
     CCC
     whatever10 ;  \    we come here, although
: EEE
     whatever11
     DDD
     whatever12 ;  \    we'd like to come here
: FFF
     whatever13
     EEE
     whatever14 ;
```

Figure 2.

```
\ {EXIT} can be used within a DO loop:
: AAA
     whatever01
     DO whatever02
        IF {EXIT} THEN  \ if we exit,
        whatever03
     LOOP
     whatever04 ;
: BBB
     whatever05
     { AAA }
     whatever06 ;       \    we come here
```

Figure 3.

Use of a return-stack marker also limits the range of the **DO** loops within which **{EXIT}** appears. In order to minimize the possibility that an intervening **DO**-loop counter will be mistaken for it, I have chosen a negative value for the marker, thus effectively limiting my **DO**-loop parameters to non-negative (or small unsigned) values.

My choice of marker was also driven by the need to distinguish between it and the more typical occupants of the return stack, addresses. Since the processors used here are byte-addressable but execute only 16-bit words, this was not difficult. I simply chose an odd marker. Since I rarely use negative numbers, or large unsigned numbers other than (even) addresses, as **DO**-loop parameters, these limitations are rarely burdensome.

The first two limitations would not exist in a system with a separate **DO**-loop stack, though other precautions would then be required, to ensure that **{EXIT}** leaves the **DO**-loop stack in a known state. (Even the standard Forth **EXIT** would have fewer limitations on such a system, but recall that **{EXIT}**'s advantages are not limited to its usability within **DO** loops.)

All three limitations could be removed by an implementation which does not put a marker on the return stack. (In order to implement **{EXIT}** on a word-addressed processor with a full complement of memory, some alternative method would, of course, be a necessity.) For example, left curly could simply store the current value of the return-stack pointer in a variable; but then only one level of curly braces could be used. Nesting of curly-brace pairs could be restored by giving left curly an array (essentially, its own stack) in which to store return-stack-pointer values, but this could be considered overkill. Perhaps a more elegant solution will be discovered. (Or one can imagine, for example, a very sophisticated **{EXIT}** that scans forward, mock-executing and doing return-stack maintenance, till it finds a **}**; **{** would then be a no-op, or could be omitted.)

Like many other pairs of Forth words, curly braces require balanced placement. Specifically, both braces of a pair must be at the same return-stack level: Either both must be outside a **DO** loop, for example, or both inside. In general, both must be in the same word (this rule may be broken with care, but both refactoring and documentation then become quite difficult). For the most part, this simply means putting the braces where they intuitively belong.

Finally, as with **EXIT**, care must be taken that **{EXIT}** leaves the parameter stack (and the **DO**-loop stack, if any) in a known condition.

Implementation Notes

Except of course for the assembler symbols in **CODE** definitions, the Forth used throughout this paper is 79-Standard. The implementation [Appendix 1] includes only two extra-Standard words, namely \ ("back-slash"), which designates the rest of the line as a comment, and **RØ** ("r-zero"), which is a user variable containing a value identical to the return-stack pointer's contents when the return stack is empty. In short, **RØ** points to the bottom of the return stack.

Ticking a colon definition is presumed to return a pointer to the first location within that definition which points to a Forth word called by the definition. (The Forth I use actually does not work this way, and the address must be calculated.)

```
\ {EXIT} can be used
\    not only in the original factoring:
: DDD
    whatever07
    whatever08
    IF {EXIT} THEN    \ if we exit,
    whatever09
    whatever10 ;
: EEE
    whatever11
    { DDD }
    whatever12 ;      \    we come here
: FFF
    whatever13
    EEE
    whatever14 ;

\ . . . but also
\    in the first refactored version:
: EEE
    whatever11
    { whatever07
      whatever08
      IF {EXIT} THEN  \ if we exit,
      whatever09
      whatever10 }
    whatever12 ;      \    we come here
: FFF
    whatever13
    EEE
    whatever14 ;

\ . . . and in the other refactorization:
: CCC
    whatever08
    IF {EXIT} THEN    \ if we exit,
    whatever09 ;
: DDD
    whatever07
    CCC
    whatever10 ;
: EEE
    whatever11
    { DDD }
    whatever12 ;      \    we come here
: FFF
    whatever13
    EEE
    whatever14 ;
```

Figure 4.

```
\ {EXIT} may be used several times
\    and/or at different levels:
: GGG
     whatever17
     IF {EXIT} THEN  \ whether we exit here,
     whatever18 ;
: HHH
     whatever19
     IF {EXIT} THEN  \    or here,
     GGG
     whatever2Ø
     IF {EXIT} THEN  \    or here,
     whatever21 ;
: III
     whatever22
     { whatever23
     IF {EXIT} THEN  \    or here,
     whatever24 }
     whatever25 ;    \    we come here
```

Figure 5.

Note that {'s run-time code, ({), compiles as two words: one is the usual pointer to ({) itself; the other word (filled in when }'s run-time code, (}), is compiled) contains the address to which {EXIT} will advance, namely the address of the word following (}).

When ({) is executed, it pushes two words onto the return stack: the advance-to address saved in its other compiled word, topped by a marker. If {EXIT} executes, it first finds the marker on the return stack (discarding it and anything on top of it), then pops the advance-to address into the Forth instruction counter, so that NEXT will advance execution to the word following (}). In this case, (}) is not executed. If {EXIT} is not executed, (}) simply discards the unused marker and address (which by then are waiting at the top of the return stack).

This implementation is coded for DEC PDP/LSI-11 computers, but an equivalent implementation of {EXIT} should be possible on most Forth systems. Register names used here are fairly typical: RP is the return-stack pointer, IC is the Forth Instruction Counter, and UV points to the current task's user variables. The op-code SEZ sets the processor-status word's Zero bit as a flag, causing EQ END to fall through. The comparison HS (high-or-same) is an unsigned greater-than-or-equal.

Acknowledgments

This work was supported by the U. S. Department of Energy Office of Inertial Fusion under agreement No. DE-FC03-85DP40200 and by the Laser Fusion Feasibility Project at the Laboratory for Laser Energetics which has the following sponsors: Empire State Electric Energy Research Corporation, New York State Energy Research and Development Authority, Ontario Hydro, and the University of Rochester. Such support does not imply endorsement of the content by any of the above parties.

References

[BROD84] Brodie, Leo, *Thinking FORTH*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

Carol Pruitt received an M.A.T. in mathematics from Harvard University in 1967, and has been programming in Forth since 1978. At the time this paper was written, she was a Senior Laboratory Engineer with the Facility Operations Group at the University of Rochester's Laboratory for Laser Energetics. Her responsibilities there included major control systems for both the single-beam Glass Development Laser and the twenty-four-beam Omega Laser. Since then, she has written ROMable code for a telephone switching system, and has begun training for a career in nutrition.

Appendix 1: Implementation

```
            Copyright © University of Rochester 1988
\ Generalized EXIT                                      23-Aug-85 CJP
-1 CONSTANT MARKER                  \ return-stack marker value
CODE ({                             \ <> - <>
              RP -)        IC )+ MOV  \ push address-after-({)
              RP -)    MARKER # MOV  \    & the marker
                                NEXT \    onto return stack
: {                                 \ <> - <address for "}" to fill>
        STATE @                     \ if we're compiling,
        IF COMPILE ({)              \    compile {'s run-time code,
           HERE                     \       & save address
           Ø ,                      \       of next word for "}"
        THEN ;   IMMEDIATE
CODE (})                            \ <> - <>
              RP )+        RP )+ CMP  \ pop marker & address
                                NEXT \    from return stack
: }                                 \ <addr of ({)'s 2nd word> - <>
        STATE @                     \ if we're compiling,
        IF COMPILE (})              \    compile }'s run-time code,
           HERE SWAP !              \    & tell ({) where (}) ends
        THEN ;   IMMEDIATE
: {ABORT}                           \ <whatever> - <>
        CR ." No {EXIT} flag"       \ let debugger know
        CR ABORT ;                  \    why we're aborting
CODE {EXIT}                         \ <> - <>
     BEGIN                                \ pop
        ' RØ @ UV I)         RP    CMP  \    return stack
        HS IF                      SEZ  \    till it's empty
        ELSE   RP )+    MARKER # CMP  \    or marker is found
        THEN
     EQ END
        ' RØ @ UV I)         RP    CMP  \ if no marker is found,
     HS IF     RP    ' RØ @ UV I) MOV  \    tidy up
               IC    ' {ABORT} #  MOV  \    & abort
     ELSE      IC          RP )+ MOV  \ else execute
     THEN                       NEXT \    the word following (})
```

```
\ NOTE:
\ A minimalist implementation would omit the STATE @ IF ... THEN
\ from { and }, would omit {ABORT}, and would define {EXIT} as
\ follows:
\
\ CODE {EXIT}                    \ <> - <>
\      BEGIN                                \ pop return stack
\              RP )+    MARKER #  CMP  \    till is marker found,
\      EQ END
\              IC           RP )+ MOV  \ then execute
\                                 NEXT \    the word following (})
```