 Table of Contents

A.9 The optional Exception word set

CATCH and THROW provide a reliable mechanism for handling exceptions, without having to propagate exception flags through multiple levels of word nesting. It is similar in spirit to the **non-local return** mechanisms of many other languages, such as C's setjmp() and longjmp(), and LISP's CATCH and THROW. In the Forth context, THROW may be described as a **multi-level** EXIT, with CATCH marking a location to which a THROW may return.

Several similar Forth **multi-level EXIT** exception-handling schemes have been described and used in past years. It is not possible to implement such a scheme using only standard words (other than CATCH and THROW), because there is no portable way to **unwind** the return stack to a predetermined place.

THROW also provides a convenient implementation technique for the standard words ABORT and ABORT", allowing an application to define, through the use of CATCH, the behavior in the event of a system ABORT.

This sample implementation of CATCH and THROW uses the non-standard words described below. They or their equivalents are available in many systems. Other implementation strategies, including directly saving the value of DEPTH, are possible if such words are not available.

SP@ (-- addr) returns the address corresponding to the top of data stack.

SP! (addr --) sets the stack pointer to addr, thus restoring the stack depth to the same depth that existed just before addr was acquired by executing SP@.

RP@ (-- addr) returns the address corresponding to the top of return stack.

RP! (addr --) sets the return stack pointer to addr, thus restoring the return stack depth to the same depth that existed just before addr was acquired by executing RP@.

```
VARIABLE HANDLER   0 HANDLER !  \ last exception handler

: CATCH  ( xt -- exception# | 0 ) \ return addr on stack
    SP@ >R          ( xt ) \ save data stack pointer
    HANDLER @ >R    ( xt ) \ and previous handler
    RP@ HANDLER !   ( xt ) \ set current handler
    EXECUTE         ( )    \ execute returns if no THROW
    R> HANDLER !    ( )    \ restore previous handler
    R> DROP         ( )    \ discard saved stack ptr
    0               ( 0 )  \ normal completion
;

: THROW  ( ??? exception# -- ??? exception# )
    ?DUP IF           ( exc# ) \ 0 THROW is no-op
    HANDLER @ RP!   ( exc# ) \ restore prev return stack
    R> HANDLER !    ( exc# ) \ restore prev handler
    R> SWAP >R      ( saved-sp ) \ exc# on return stack
    SP! DROP R>     ( exc# ) \ restore stack
        \  Return to the caller of CATCH because return
        \  stack is restored to the state that existed
        \  when CATCH began execution
THEN
;
```

In a multi-tasking system, the HANDLER variable should be in the per-task variable area (i.e., a user variable).

This sample implementation does not explicitly handle the case in which CATCH has never been called (i.e., the ABORT behavior). One solution is to add the following code after the IF in THROW:

```
        HANDLER @ 0= IF ( empty the stack ) QUIT THEN
```

Another solution is to execute CATCH within QUIT, so that there is always an **exception handler of last resort** present. For example:

```
: QUIT      ( empty the return stack and )
            ( set the input source to the user input device )
    POSTPONE [
    BEGIN
      REFILL
    WHILE
      ['] INTERPRET CATCH
      CASE
      0 OF STATE @ 0= IF ." OK" THEN CR  ENDOF
     -1 OF ( Aborted) ENDOF
     -2 OF ( display  message from ABORT" ) ENDOF
      ( default ) DUP ." Exception # "  .
      ENDCASE
    REPEAT BYE
;
```

This example assumes the existance of a system-implementation word INTERPRET that embodies the text interpreter semantics described in 3.4 The Forth text interpreter. Note that this implementation of QUIT automatically handles the emptying of the stack and return stack, due to THROW's inherent restoration of the data and return stacks. Given this definition of QUIT, it's easy to define:

```
        : ABORT  -1 THROW ;
```

In systems with other stacks in addition to the data and return stacks, the implementation of CATCH and THROW must save and restore those stack pointers as well. Such an **extended version** can be built on top of this basic implementation. For example, with another stack pointer accessed with FP@ and FP! only CATCH needs to be redefined:

```
: CATCH  ( xt -- exception# | 0 )
    FP@ >R  CATCH  R> OVER IF FP! ELSE DROP THEN ;
```

No change to THROW is necessary in this case. Note that, as with all redefinitions, the redefined version of CATCH will only be available to definitions compiled after the redefinition of CATCH.

CATCH and THROW provide a convenient way for an implementation to **clean up** the state of open files if an exception occurs during the text interpretation of a file with INCLUDE-FILE. The implementation of INCLUDE-FILE may guard (with CATCH) the word that performs the text interpretation, and if CATCH returns an exception code, the file may be closed and the exception reTHROWn so that the files being included at an outer nesting level may be closed also. Note that the Standard allows, but does not require, INCLUDE-FILE to close its open files if an exception occurs. However, it does require INCLUDE-FILE to unnest the input source specification if an exception is THROWn.

A.9.3 Additional usage requirements

One important use of an exception handler is to maintain program control under many conditions which ABORT. This is practicable only if a range of codes is reserved. Note that an application may overload many standard words in such a way as to THROW ambiguous conditions not normally THROWn by a particular system.

A.9.3.6 Exception handling

The method of accomplishing this coupling is implementation dependent. For example, LOAD could **know** about CATCH and THROW (by using CATCH itself, for example), or CATCH and THROW could **know** about LOAD (by maintaining input source nesting information in a data structure known to THROW, for example). Under these circumstances it is not possible for a Standard Program to define words such as LOAD in a completely portable way.

A.9.6 Glossary

A.9.6.1.2275 THROW

If THROW is executed with a non zero argument, the effect is as if the corresponding CATCH had returned it. In that case, the stack depth is the same as it was just before CATCH began execution. The values of the i*x stack arguments could have been modified arbitrarily during the execution of xt. In general, nothing useful may be done with those stack items, but since their number is known (because the stack depth is deterministic), the application may DROP them to return to a predictable stack state.

Typical use:

```
: could-fail ( -- char )
    KEY DUP [CHAR] Q =  IF  1 THROW THEN ;

: do-it ( a b -- c)   2DROP could-fail ;

: try-it ( --)
    1 2 ['] do-it  CATCH  IF ( x1 x2 )
        2DROP ." There was an exception" CR
    ELSE ." The character was " EMIT CR
    THEN
;

: retry-it ( -- )
    BEGIN  1 2 ['] do-it CATCH  WHILE
       ( x1 x2) 2DROP  ." Exception, keep trying" CR
    REPEAT ( char )
    ." The character was " EMIT CR
;
```

 Table of Contents

 Next Section