CATCH and THROW

Michael Milendorf, Sun Microsystems, Inc.

Michael.Milendorf@Sun.Com

Abstract

The CATCH and THROW exception handling mechanism (non-local or multilevel exit) is one of the most abstruse fundamental concepts to have been added to the Forth programming language as originally defined by C. Moore. At the same time the idea behind, and implementation of this mechanism is very elegant, and intrinsic to Forth. This mechanism is also easy to use if properly understood. This paper describes the history behind the discovery of CATCH and THROW, their implementation, syntax, semantics, and innovative methods to use them effectively in complex multi-layered Forth systems such as OpenFirmware.

Acknowledgments

Special thanks go to William Mitch Bradley of FirmWorks for sharing the story of his involvement in **CATCH** and **THROW**, and for allowing its inclusion in this papers. Also thanks to Paul Thomas of Sun Microsystems for his feedback and interpretation of ANS Forth definitions of **ABORT** and **ABORT"** and to Robert Houk of Sun Microsystems for thorough technical review of the paper.

1 History

When Forth was invented, the handling of exceptions was out of the scope of the programming language. The exception handling mechanism as defined by ANS Forth was suggested by William Mitch Bradley in 1990. At that time Bradley led an OpenBoot firmware development effort at Sun Microsystems, Inc. and also served as a vice chairman of the X3J14 ANS Forth Technical Committee.

Before he came up with the idea, Bradley studied all the other Forth exception handling schemes that had been published at the time. He also studied schemes from several other languages (C, Modula, Cedar, LISP, and PostScript). The other-language exception handling was stylistically consistent with those languages. For example, Cedar's exception handling was tied in with the language syntax, and required a fair amount of compiler support, while C's **setjmp()**/**longjmp()** required no additional support from the compiler from a syntax point of view, but depended on the linker to resolve the jump buffer address, which was effectively a global variable. Both of these were consistent with the overall flavor of those languages.

Bradley wrote, that it finally occurred to him that the problem with all of the other Forth exception-handling schemes was, that they weren't very Forth-like. They tended to depend on some combination of syntactic elements (prefix operators, defining words, built-in control structures) or data structures. He realized that the ideal solution would be to build the mechanism around an exposed primitive that was basically just an ordinary non-immediate, non-defining, stack-based Forth word.

Bradley well remembered that the place where the concept finally crystallized in his mind was the bathtub in a hotel room. It was on the morning of an ANS Forth technical committee meeting that was being held at the NASA Goddard facility at College Park, Maryland in 1990. Later that day Bradley broached the subject of exception handling, which was one of the committee's sore spots at the time. Everybody knew that the standard needed to address the issue, but nobody really liked any specific proposal well enough to get behind it and push. The problem with complicated schemes, which always surfaces in committee discussion, was that they tend to have problematic interactions with other aspects of Forth. In such cases, the discussions tend to get off the track and eventually stall.

Later on Bradley worked out the details of the implementation and was gratified at how simple it turned out. He started using it in various places in the OpenBoot source base, and found out through experience that it worked really well. He was especially pleased at how flexible and scalable the **CATCH** and **THROW** scheme turned out to be, but he was pretty much expecting it to turn out that way, because that is the key to Forth - avoiding syntactic complexity results in maximum flexibility. After the positive experiences using **CATCH** and **THROW** in OpenBoot, which is a large wide-ranging application that must be reliable, Bradley proposed his **CATCH** and **THROW** model to the ANS Forth committee. As he recalls, it was accepted fairly easily, as such things go, with less that the usual amount of bickering, wailing, and gnashing of teeth. About the only change to Bradley original scheme that they made in the committee was to add the **0 THROW** is a **noop** stipulation. That was done when the committee considered the use of **THROW** after words that return an **ior** (implementation-defined I/O result code). Phrases like **OPEN-FILE THROW** are very satisfying. Quite a while later, the committee finally got around to assigning some specific throw values for standard error conditions.

Bradley borrowed the names **CATCH** and **THROW** from LISP, which is the language whose exception handling is the most similar to the ANS Forth **CATCH** and **THROW**. LISP's **catch** and **throw** are tagged, which means that if you say **(throw foo)**, it will only trigger a **catch** that specifies the same tag **foo**. Bradley contemplated doing something like this with **CATCH** and **THROW** in Forth, in which a given **CATCH** would only respond to particular throw values, but he decided against it. He wrote some example code sequences using the different semantics, and found that the “catch everything and re-throw if needed” semantics yielded cleaner code in nearly every case, especially in conjunction with **CASE**. Since then, **CATCH** and **THROW** have been continued to be extensively used across OpenBoot source base. OpenBoot uses the same original definitions of **CATCH** and **THROW** as Bradley wrote in 1990. **CATCH** and **THROW** are also standard FCodes. These FCodes are being increasingly used to code firmware diagnostic solutions for I/O devices and FCode drivers for Sun Microsystems platforms.

2 Syntax and Semantics

In essence, **THROW** is an abort (multilevel exit) and **CATCH** is an “abort-proof” version of **EXECUTE**. If a word is executed with **CATCH**, and a **THROW** occurs during the execution of that word, the return stack is unwound, the data stack is cleaned up, and controls returns to the **CATCH** point. If **CATCH** had not been used, the **THROW** would have unwound all the way to the top level, and the program would have been blown away without getting a chance to participate in the recovery process. ANS Forth presents **CATCH** and **THROW** as a part of THE OPTIONAL EXCEPTION WORD SET:

9.6.1.0875 CATCH (i*x xt -- j*x 0 | i*x n)

Casual definition: Execute command indicated by ***xt***. Return throw result ***n***.

ANS Forth: *Push an exception frame on the exception stack and then execute the execution token* ***xt*** *(as with* ***EXECUTE****) in such a way that control can be transferred to a point just after* ***CATCH*** *if* ***THROW*** *is executed during the execution of* ***xt****.*

9.6.1.2275 THROW (k*x n -- k*x | i*x n)

Casual definition: Transfer back to **CATCH** routine if return code ***n*** is non-zero.

ANS Forth definition: *If any bits of* ***n*** *are non-zero, pop the topmost exception frame from the exception stack, along with everything on the return stack above that frame. Then restore the input source specification in use before the corresponding* ***CATCH*** *and adjust the depths of all stacks defined by the standard so that they are the same as the depths saved in the exception frame (****i*** *is the same number as the* ***i*** *in the input arguments to the corresponding* ***CATCH****), put* ***n*** *on top of the data stack, and transfer control to a point just after the* ***CATCH*** *that pushed that exception frame. If the top of the stack is non-zero and there is no exception frame on the exception stack, the behavior is as follows: If* ***n*** *is minus-one (-1), perform the function of* ***ABORT****. If* ***n*** *is minus-two (-2), perform the function of* ***ABORT”****. Otherwise the system may display an implementation-dependent message giving information about the condition associated with the throw code* ***n****. Subsequently the system shall perform the function of* ***ABORT****.*

3 Implementation of CATCH and THROW

Bradley’s implementation of **CATCH** and **THROW** is very simple when understood.

CATCH saves the data stack pointer, the address of the nearest enclosing other **CATCH** frame on the return stack, and saves the address of the return stack frame it just created in the global variable *handler*. It then executes the word denoted by it’s execution token argument. If that word completes normally (without **THROW**ing), control returns to **CATCH** via the normal Forth unnesting mechanism, **CATCH** removes the stuff that it put on the return stack, restores the *handler* variable to its previous contents, pushes a zero on the data stack, and returns to its caller.

If, on the other hand, **THROW** is executed (with a non-zero argument), **THROW** locates the nearest enclosing **CATCH** frame (whose address is in *handler*), cleans off the return stack down to and including that **CATCH** frame, restores the data stack pointer to the value saved in that frame, pushes the throw value on the stack, and returns to **CATCH**’s caller.

When **CATCH** returns zero (meaning that **0 THROW** was executed, or that no **THROW** was executed at all), the state of the stack underneath that zero is exactly as it would have been had **CATCH**'s argument been executed with **EXECUTE**. When **CATCH** returns a non-zero value (meaning that **THROW** was executed with a non-zero value), the depth of the stack, not counting the throw value itself, is the same as it was just before **CATCH**'s argument word was executed. It should be emphasized that it is only the depth of the stack that is preserved, not the contents of the stack (more literally, the data stack pointer itself is "restored" without regard to the contents of the stack).

While ANS Forth doesn't require use of the return stack for implementation of **CATCH** and **THROW**, the use of the return stack comes naturally and implemented in most systems. The suggested implementation of **CATH** and **THROW** was originally written by Mitch Bradley and currently used in all FirmWorks' OpenFirmware and Sun Microsystems' OpenBoot systems. The implementation uses the set of non-standard words described below. These words or their equivalents are available in many systems:

```
sp@ ( -- addr ) returns the address corresponding to the top of data stack.
rp@ ( -- addr ) returns the address corresponding to the top of return 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 -- ) 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               \ Most recent exception handler

: CATCH  ( xt -- exception# | 0) \ Return address is already on the stack
   sp@ >r           ( xt)        \ Save data stack pointer
   handler @ >r     ( xt)        \ Save previous handler
   rp@ handler !    ( xt)        \ Set current handler to this one
   execute          ( )          \ Execute the word passed in on the stack
   r> handler !     ( )          \ Restore previous handler
   r> drop          ( )          \ Discard saved stack pointer
   0                ( 0)         \ Signify normal completion
;

: THROW  ( ??? exception# -- ??? exception# )  \ Returns in saved context
   dup  0=  if  drop exit  then  \ Don't throw 0
   handler @ rp!    ( exc#)      \ Return to saved return stack context
   r> handler !     ( exc#)      \ Restore previous handler
                    ( exc#)      \ Remember exc# on return stack
                    ( exc#)      \ before changing data stack pointer
   r> swap >r       ( saved-sp)  \ exc# is on return stack
   sp! drop r>      ( exc#)      \ Change stack pointer
   \ This return will return to the caller of CATCH, because the return
   \ stack has been restored to the state that existed when CATCH began
   \ execution.
;
```

4 Throw Values

Throw values must only be selected from ranges as defined by ANS Forth to avoid conflicts between the Forth system and application programs. Throw values {-255 ... -1} shall be used only as assigned by ANS Forth. The values {-4095 ... -256} shall be used only as assigned by a system. Application programs shall not define values for use with **THROW** in the range {-4095 ... -1}. ANS Forth provides (Table 9.2, page 71) throw values in the range {-58 ... -1} assigned to different error or exception events (-1 is reserved to the **ABORT** function, -2 is reserved to **ABORT"**, -3 to *stack overflow*, -4 to *stack underflow*, etc.).

5 ABORT and ABORT"

ABORT and **ABORT"** are words which existed in the Forth programming language before the **CATCH** and **THROW** mechanism was discovered in 1990. **ABORT** can be casually described as a multilevel exit, while **ABORT"** is a multilevel exit with an associated message. The syntax and semantics for those ANS Forth CORE words are defined as:

6.1.0670 ABORT (i*x --) (R: j*x --)

ANS Forth: *Empty the data stack and perform the function of* ***QUIT****, which includes emptying the return stack, without displaying a message.*

6.1.0680 ABORT" (i*x flag -- | i*x) (R: j*x -- |j*x) Compilation: **(" ccc" --)**

ANS Forth: *Remove* ***flag*** *from the stack. If any bit of* ***flag*** *is non-zero, display* ***ccc*** *string and perform an implementation-defined abort sequence that includes the function of* ***ABORT****.*

In addition, ANS Forth defines two other versions of **ABORT** and **ABORT"** for systems who choose to include the **CATCH** and **THROW** functions in their implementation. Those "more powerful" versions of **ABORT** and **ABORT"** are presented as a part of the OPTIONAL EXCEPTION WORDS SET:

9.6.2.0670 ABORT (i*x --) (R: j*x --)

ANS Forth: *Extend the semantics of* ***6.1.0670 ABORT*** *to be: Perform the function of* ***-1 THROW****.*

We must admit, that ANS Forth is not being very clear in definition of **9.6.2.0670 ABORT**. What is really meant is that **9.6.2.0670 ABORT** *replaces*, not *extends* the semantics of **6.1.0670 ABORT** to **-1 THROW**. ANS Forth defines **9.6.2.0670 ABORT** to simplify error recovery. Using **CATCH** and **THROW** allows the program to make more intelligent decisions as to how to proceed when an error is encountered. It also allows different behaviors depending on whether or not the **ABORT** has a **CATCH**er.

9.6.2.0680 ABORT" (i*x flag -- | i*x) (R: j*x -- |j*x) Compilation: **(" ccc" --)**

ANS Forth: *Remove* ***flag*** *from the stack. If any bit of* ***flag*** *is not zero, perform the function of* ***-2 THROW****, displaying* ***ccc*** *string if there is no exception frame on the exception stack.*

ANS Forth defines **9.6.2.0680 ABORT"** because programmers want to use **ABORT"** but do not always want the text to be displayed, especially for embedded applications where there is not always a display present. For debugging or larger systems, the message could come out. Or the message could be saved to a log. Or it could be displayed at a later time. Or it could be translated into another language and then displayed. The ANS Forth committee decided that using **CATCH** and **THROW** would allow intervention in the display of the message, and liked the idea.

In regards to the **9.6.2.0680 ABORT”** function, three distinct behaviors of the function could be observed, depending on the implementation of **ABORT”** itself and on the **CATCH**er: (**1**) **ABORT”** displays the text first, then executes **-2 THROW**; (**2**) **ABORT”** saves the message in the buffer, then executes **-2 THROW**; **CATCH**er displays the text; (**3**) **ABORT”** saves the message in the buffer, then executes **-2 THROW**; **CATCH**er doesn’t display the text.

6 Effective use of CATCH and THROW in a program

As was mentioned earlier, Bradley contemplated making **CATCH** only respond to particular throw values, but he decided against it. He found that the “catch everything and re-throw if needed” semantics yielded cleaner code in nearly every case, especially in conjunction with the **CASE** statement. These semantics must be clearly understood. Remember that **CATCH** is always catching all **THROW**s, initiated by an application program as well as by the Forth system. Therefore, catching and not re-throwing a “system” throw value may alter the intended execution flow of a Forth system, in case of a failure or exception. In addition, interception of “system” **THROW**s (or **ABORT”**s) will make some Forth systems not display an intended error message:

```
: foo ( -- )  ... 10 THROW ;         \ Somewhere in the application
['] foo CATCH  ?dup  if  ...  then   \ “sloppy” CATCHer
```

While this example may look correct, in reality it is not a foolproof coding. The assumption is being made that the only **THROW** which will be **CATCH**ed is the one originating in the **foo** definition, and throwing value **10**. With this assumption, the program just checks if the throw value is non-zero, and assumes that if it’s non-zero, it is the **10** value thrown from within **foo**. Wrong! What if, during execution of **foo**, the Forth system generates a “system” throw with one of “reserved” throw codes? In such case, a “system” **THROW** initiated by the system (for example, a *stack overflow* or -3 throw code) will be intercepted (**CATCH**ed) in the application program, and the program execution will continue from the point of **CATCH** (with a possibly corrupted data stack), instead of exiting to the top level and printing an error message, as was originally meant by the Forth system executing a “system” **THROW**.

Taking this into consideration, the correct solution would be to check the throw value **CATCH**ed explicitly against every throw value known to the application program. Then re-**THROW** (propagate) any unknown to the application program (system) codes, to let the Forth system take proper care of any “system” throws at the top level:

```
['] foo CATCH dup 10 =  if  ( 10|n) \ check the throw code = 10
   ...                      ( 10)   \ known code, proceed
else                        ( n)    \ otherwise
   THROW                    ( )     \ re-THROW unknow “system” code;
then                                \ 0 THROW executes noop
```

If more than one error code must be handled, the best way is to use the **CASE** statement in the following way:

```
['] foo CATCH
case
   10  of  ....   endof             \ respond to THROW CODE = 10
   20  of  ....   endof             \ respond to THROW CODE = 20
   40  of  ....   endof             \ respond to THROW CODE = 40
         dup THROW                  \ re-THROW unknown "system" ;
endcase                             \ 0 THROW executes noop
```

With this suggested programming technique, the use of **CATCH** in the application program will never affect the intended behavior of the Forth system, in case a "system" **THROW** (or **ABORT**) is executed.

Another requirement for the effective use of **CATCH** and **THROW** is to never use, in an application program, throw values reserved for system use. In most cases an application program shall use only small positive numbers for throw values, exactly as specified by ANS Forth.

An interesting variation for using **THROW** is to employ a memory address as a throw value. This memory address may contain a packed string (text message) corresponding to the error, triggered by the **THROW**:

```
ok : foo ( -- ) ... p" XXX" THROW ;
ok ['] foo CATCH ( paddr ) .error
ok XXX
```

In this example, the throw value returned by **CATCH** (*paddr*) is the address of the packed string, containing the message "XXX". The **.error** word determines if the throw value could be interpreted as a valid memory address, and executes **COUNT TYPE** which effectively prints the XXX message on the console.

A number of different *types* of throw values can be used in a Forth program: (1) positive error codes; (2) positive error codes to be selected by the **CASE** statement; (3) small positive contiguous number range used as an idex in a dispatch (or message) table; (4) memory addresses, containing packed strings of error messages, which can be displayed by a **CATCH**er; (5) ***xt*** or any other memory addresses.

Another observation is that because **CATCH** and **THROW** adjust both return and data stacks in most systems, it is not necessary to clean data and return stacks prior to execution of **THROW:**

```
ok : foo ( -- ) 1 >r 2 >r 3 4 5 THROW ;   (   )
ok ['] foo CATCH                          ( 5 )
```

It's often an assumption that the **CATCH** and **THROW** mechanism could only be used for error handling cases. While surely error handling is the primary application of the exception handling mechanism, **CATCH** and **THROW** could be also used for effective programming solutions, when multilevel exit makes programs simpler.

7 Summary

While numerous exception handling mechanisms were suggested before and after discovery of **CATCH** and **THROW**, it is our belief that this mechanism is of a fundamental nature to the Forth engine. **CATCH** and **THROW** have proven effective in a variety of Forth systems and applications (such as OpenFirmware and OpenBoot and FCode drivers). It is important to follow ANS Forth when picking throw values in a program, and code every **CATCH** statement properly, remembering that “system” throws along with application program throws may be **CATCH**ed by almost any **CATCH** in the program.

In regards to aborting a program, ANS Forth defines two sets of **ABORT** and **ABORT”** functions: One is a part of THE CORE WORD SET (which doesn’t use **THROW**) and another one is a part of THE OPTIONAL EXCEPTION WORD SET (which does use **THROW**). The last set allow powerful and flexible error handing by using **CATCH** to control the behavior of **ABORT** and **ABORT”**.

8 References

[**1**] ANSI X3.215-1994 ANS for Information Systems - Programming Languages - Forth

[**2**] IEEE Standard 1275-1994; Standard for Boot Firmware: Core Requirements & Practices

[**3**] Gassanenko, M.L., Extension of the Exception Handling Mechanism, Euroforth95

[**4**] Rodriguez, B.J., A Forth Extension Handler, SIGForth, Vol. 1, Summer’89, p. 11-13

[**5**] Wejgaard, W., TRY: A Simple Exception Handler, EuroFORML ‘91, p.4

[**6**] Clifton, G., Terry, R., Exception Handling in Forth, Rochester Forth Conference ‘85

[**7**] Woehr, J., Forth: The New Model, M&T Books, 1992