Technical Note

Nested Error Handlers

Bill Stoddart
Department of Computer Science
Teeside Polytechnic
Middlesborough, Cleveland, U.K.

Abstract

As code executes, an application's error handlers may be added to or removed from a sequence that will be executed before entering a system error handler. These user defined error handlers can ensure that an aborting task has a minimum impact on the rest of the system, and that it is correctly re-initialised for subsequent use.

The Problem

When Forth detects an error it executes an error handler such as the run time operator associated with the 83-Standard word `ABORT"`.

The typical action of this error handler is to report the error condition, clear the stacks, and perform whatever system dependent abort sequence is needed to restore the executing task to a safe condition. Typically the operator task will be restored to a state in which it will accept and interpret subsequent keyboard input, and other terminal tasks will be restored to a state in which they can accept messages from other tasks. (Definitions of terms used are included in an appendix).

In some applications however, a special error handler is required to leave the system in a viable condition. Consider a task whose output has been vectored to a disc file. We might wish to specify that its output is vectored back to the console on detecting an error.

Providing vectored error handling is not enough to solve this problem, as can be seen from the following example. Suppose the operator task vectors its console input to come from a file and specifies an error handler that will vector console input back to a keyboard. Then suppose the operator task vectors its output to a printer and sets an error handler that will vector console output back to the screen. If an error now occurs we need *both* the error handlers to be executed.

A Solution

The word `ON-ERR` is used within a high level Forth definition in the form ...`ON-ERR` <handler> ... where <handler> is a user defined error handler.

Each terminal task may specify up to 4 such handlers. When an error is encountered these will be executed before entering the task's abort sequence. The most recently specified handler is executed first.

The word `-ON-ERR` deletes the most recently specified error handler from the sequence.

`RECONFIG` will execute the sequence of user specified error handlers. `RECONFIG` is invoked by the system error handlers, such as the run time operator compiled by `ABORT"`.

Implementation Notes

A user variable ESP is used as an error stack pointer. The 5 cells that follow ESP in the user area are reserved for the error stack. The 5th cell is initialised to contain the execution address of EXIT, and ESP is initialised to point to this cell. The remaining 4 cells are used to support up to 4 levels of user specified error handling.

When a word containing ...ON-ERR <handler>... is compiled, the execution address of <handler> will be compiled in line following the execution address of ON-ERR. When ON-ERR executes, the address of the cell into which <handler> has been compiled will be on top of the return stack.

ON-ERR fetches the top of the return stack to locate <handler>. It must also increment the value fetched by 2 and replace it on the return stack so that the word following <handler> will be executed on exit from ON-ERR. This is common technique, used by other Forth words such as COMPILE.

The address of <handler> is pushed onto the error stack. A check is made to ensure that the error stack cannot overflow. If the error stack is full, a warning is given and the address of <handler> is not added to the stack.

-ON-ERR removes an item from the error stack unless the error stack is already empty, in which case a warning is issued.

RECONFIG places a pointer to the top of the error stack on the return stack. When RECONFIG exits, control passes to the top of the error stack, and the error handlers on the stack are executed. A second exit occurs when execution reaches EXIT at the bottom of the error stack. This returns to the word following RECONFIG and leaves the return stack in a balanced condition.

Source Code

```
: ON-ERR
  R> DUP 2+ >R ( fetch top of return stack and update it )
  ESP @  ESP 2+ = (error stack overflow check )
  IF   3 ?WARN ( issue a warning if overflow is detected )
  ELSE  @   -2 ESP +!   ESP @ ! ( add <handler> to error stack )
  THEN  ;

: -ON-ERR
  ESP @  ESP 10 + = ( error stack underflow check )
  IF TRUE 4 ?WARN ( warning if underflow is detected )
  ELSE  2 ESP +! ( remove item from error stack )
  THEN  ;

: RECONFIG  ESP @  >R  ;
```

Reflections

A distinction is often made between recoverable and unrecoverable errors. The error handlers discussed here deal with unrecoverable errors. Something has occurred which makes a task unable to sensibly continue execution. The following considerations then apply.

1. The task must abort in a manner that has the minimum impact on other tasks in the system. For example it must release any shared resources that it currently owns. Forth has no centralised operating system kernal which keeps track of resources owned by all tasks. Its solution to shared facilities is far simpler, and requires tasks to get and release facilities via operations performed on facility variables. Nested error handlers make this simple to achieve under error abort conditions.

2. The task must return to a condition in which it can be used again. For example the operator task will clear its stacks and await further keyboard input. Other tasks will typically clear their stacks and wait to be reactivated by a message from another task.

3. The error must be reported to a higher level so that activities can be replanned. With error handlers such as `ABORT"` the higher level is simply the human user of the system, but as AI techniques become more widely used in real time applications "error reports" will increasingly be made to a task planning level of software. This will blur the distinction between recoverable and unrecoverable errors, and lead to a greater need for carefully planned error handling techniques.

Acknowledgements

The multitasking terminology and the techniques used to share facilities between tasks originated with polyFORTH™.

Appendix

Definition of terms and non-standard words.

`?WARN` flag n --
Issue warning n if flag is true, then continue execution.

facility variable
A variable used to share a system facility between tasks. For example we might define `VARIABLE PRINTER`. When the printer is free for use `PRINTER` will contain 0. Each task that uses the printer executes `PRINTER GET` before doing so. If the printer is owned by another task `GET` polls the facility variable until the facility is free. It then places its own task address in the facility variable and proceeds. The phrase `PRINTER RELEASE` must subsequently be executed to return the printer to a free condition. `RELEASE` checks that the executing task actually owns the facility and if so writes a 0 into the facility variable. This means that `RELEASE` is safe to use in an error handler without checking whether the facilities in question are currently owned by the executing task.

operator task
A task present at cold start and which handles Forth's interaction with the system console.

terminal task
A task that has the same software capabilities as the operator task. A terminal task could be configured to support its own hardware console, but for the present discussion may be thought of as interpreting Forth source code placed in its terminal input buffer by other tasks.

83-Standard Notes

Since the Standard does not specify the form in which return addresses are held, `ON-ERR` and `RECONFIG` cannot be considered as being written in Standard Forth. For example a Forth implementation with a pre-incrementing inner interpreter would need slightly different versions.

Bill Stoddart received a BSc in Mathematics and an MSc in Statistics from Sheffield University U.K. He has worked as a Systems Engineer and a Yoga teacher, and is now a Senior Lecturer in the Dept. of Computer Science at Teeside Polytechnic, Cleveland, U.K. He has implemented a multitasking 83-Standard Forth which is used for teaching and projects in the Polytechnics Real Time Laboratory, and is currently working with Universal Machine Intelligence, Ltd. on the use of Forth in robotics.