EuroForth 2011

Crash Never

N.J. Nelson B.Sc. C. Eng. M.I.E.T.
Micross Automation Systems
4-5 Great Western Court
Ross-on-Wye
Herefordshire
HR9 7XP
UK
Tel. +44 1989 768080
Email njn@micross.co.uk

Abstract
Two approaches are contrasted in the search for reliability of a large and complex Forth system.

1. Introduction

In our terms, the reliability of a program means that it continues to work without stopping, for a very long period - at least months and maybe years.
Two possible approaches can be imagined for obtaining software reliability:
a) "Crash early, crash often"
You make a system that is extremely intolerant of programming errors, and thus try to force them to reveal themselves at an early stage.
b) "Crash never"
You accept that you can never be a perfect programmer. You make a system which tries to struggle on despite programming errors.
It is a combination of these approaches which we have been using (mostly by accident) for many years.

2. Crash early, crash often

This approach is best suited to applications which can be thoroughly and systematically tested. Every possible program path can be simulated, and hopefully a wide variety of potential data.
This does not always work, no matter how many resources are put into the testing process. It has been claimed that Bill Gates managed to "blue screen" every new version of Windows in front of a live audience.

3. Crash never

This is perhaps better labelled "Don't crash in front of the customer". This approach is better for systems for which thorough testing is not an economic possibility. In our case, it's not economic because each copy of our software is different, and is sold only once. And the reason it is different is that it is controlling a set of mechanical equipment which is unique to each customer.

4. Why do programs crash?

First, we need to set aside logical errors. For example, the customer may inform you "every time I press this button this happens, instead of that". These are usually not programming errors, but errors of specification (the mechanical engineer explained it wrongly to the programmer). Logical errors are usually easy to fix. By crash I mean that all or part of the program stops working. Typical causes for this are:

a) Invalid address errors
b) Huge, or infinite loops
c) Division by zero, or overflow
d) Race, or gridlock conditions

5. Invalid address errors, and what to do about them

Most invalid addresses are caused by stack errors. In a Forth environment, the compiler does not check the number and type of parameters which are passed to and from a word (function). This leads to the possibility of accumulating stack errors, even if there is no logical error in the function. Depending on how frequently a function is called, this may result in an invalid address either immediately, or at some time in the future.

```
: DIE-IMMEDIATELY
  1000000 0 DO DROP ( or DUP, if you fancy ) LOOP
;

: DIE-IN-A-MONTH
  BEGIN DROP 1000000 WAIT AGAIN
;
```

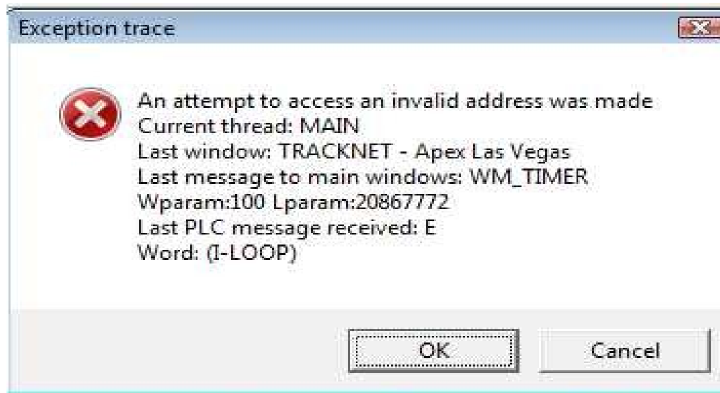
In a Windows program, the majority of the code handles responses to messages that Windows sends you - for example when a key is pressed or the mouse is moved. It so happens that the compiler that we have been using for many years (MPE ProForth V2.1) creates a new stack, every time a new message is processed. Quite by accident, an extremely fault-tolerant system is created - the system is largely immune to the most common programming error! It is due to this one factor (and certainly not to our own programming expertise) that we have a reputation for producing highly reliable software.

Of course, more sophisticated Windows programs consist of more than reponses to Windows messages. Additional threads of execution (TASKs in Forth) are created, to carry out background processing. These tasks essentially consist of infinite loops, called at predetermined intervals, and are therefor prone to die-in-a-month syndrome. Our solution is to create a stack guardband.

```
: SETDEPTH ( n--- ? ) \ Set stack depth as protection against under/overrun
  DEBUGGING 0= TURNKEY? OR    \ Always protect if in turnkey mode
  STACKPROTECT @ OR IF        \ And in debug mode, if stack protection on
    BEGIN
      DUP 1+ DEPTH <>         \ Depth not as required
    WHILE
      DUP DEPTH < IF          \ Too much
        NIP                   \ Down a bit
      ELSE                    \ Too little
        0 SWAP                \ Up a bit
      THEN
    REPEAT
  ELSE
    DROP                      \ The requested depth
  THEN
;

TASK: MYTASK( --- ) \ My processing task
  BEGIN
    10 SETDEPTH         \ Guard against stack errors
    50 WAIT             \ Do it every 50ms
    DO-SOME-WORK        \ The background work
  AGAIN
;
```

Although by far the majority of invalid address errors are caused by stack mistakes, they can also be caused by other factors, such as the miscalculation of an index into an array. There is nothing we can do to immunise the system against these, so the response is to indicate in as much detail as possible the location of such an error, whenever it occurs.



The message box describes

- the type of exception
- the thread in which it occurred
- details of the last window message which was started
- special information about a particularly error-prone communications function
- the name of the Forth word which caused the exception.

This is a very valuable tool. The code required to produce this information is complex, and in this paper, there is only time to examine the outmost word. Any delegate is welcome to see the full source if they are interested. Note that the detail is highly compiler dependent, and adjustments would be needed for other compilers.

```
: (EXCEPTION-HANDLER) { | esi -- } \ Display exception message box
  WINAPPHANDLE@                                \ Owner
  LOAD-ADDR @ @                                \ Get exception code
  GET-EXCEPTION-STRING $>ASCIIZ                \ Convert to string
  ZCRLF Z+ Z"" Current thread: " Z+            \ Show name of thread
  SELF ZTHREAD Z+
  ZCRLF Z+ Z"" Last window: " Z+               \ Show name of last window
  CURR-WINDOW HANDLE ZWINNAME Z+
  ZCRLF Z+ Z"" Last message to main windows: " \ Show last main message
  Z+ LASTWM @ WMNAME Z+
  LASTWM @ WM_COMMAND = IF                     \ It was a command message
    ZCRLF Z+ Z"" Command: " Z+                 \ Show command
    LASTCM @ ZTEXT Z+
  THEN
  ZCRLF Z+    Z"" Wparam:" Z+ LASTWPARAM @     \ Last w l params to main
  ZFORMAT Z+ Z""  Lparam:" Z+ LASTLPARAM @
  ZFORMAT Z+
  ZCRLF Z+ Z"" Last PLC message received: " Z+ \ Show last PLC message
  LASTPLCMESSAGE Z+
  ZCRLF Z+ Z"" Word: " Z+                      \ Attempt to identify Forth
word
  LOAD-ADDR @ 4 + @ ABS>REL                    \ Get base of context
structure
  112 +                                        \ Offset for
floating_save_area
  12 cells+ @ -> esi                           \ Get Esi
  esi FORTH-BASE - IDENTIFY-IP Z+
  ZCRLF Z+ TRACESTRING Z+                      \ Concatenate trace
information
  Z"" Exception trace"
  MB_APPLMODAL MB_ICONHAND or MB_OKCANCEL OR
  WINMESSAGEBOX IDCANCEL = IF                  \ Display box, did user
cancel?
    BYE
  THEN
  TRACESTRING OFF                              \ Clear trace
  LOAD-ADDR @ OFF                              \ Reset so cold will work OK
  ABORT                                        \ Warm restart
; ASSIGN (EXCEPTION-HANDLER) TO-DO EXCEPTION-HANDLER
```

6. Huge, or infinite loops, and what to do about them

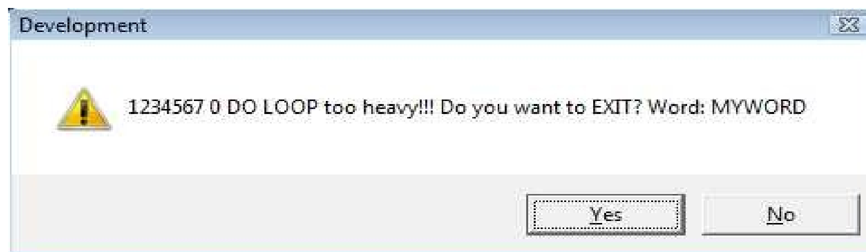
There are two types of loop errors:

a) DO..LOOPs with miscalculated input parameters

b) BEGIN.. with miscalculated WHILE or UNTIL parameters

6a. Miscalculated DO.. LOOPs

Modern PCs are rather speedy, and can process very large loops without noticeable delay. Nevertheless, if a DO.. LOOP is asked to execute, say, a million times, then it is reasonable to assume there may be something wrong.



The word which contains the offending loop is shown. A definition of the words DO, ?DO and LOOP is required to achieve this, and again code is available to anyone interested.

6b. BEGIN.. with miscalculated WHILE or UNTIL parameters

The effect of these, when included in a Windows message, is that the system becomes unresponsive to the mouse and keyboard. If such a problem happens very infrequently, it can be extremely hard to find. Fortunately, although our programs run continuously for very long periods, in practice there is only an operator interacting with the system for a small percentage of that time. If we were able to detect a non-responsive program, and close then restart it automatically, then if we're lucky the customer might not notice! At the very least, we may buy some time to locate the error.

This is one of those occasions when age is an advantage. We can remember writing code for early primitive microprocessors, which were very prone to disruption caused by electromagnetic incompatibility. We therefore "invented" (contemporaneously no doubt with many others) a simple and foolproof external circuit which, if not reset regularly by the software, would in turn reset the microprocessor. We called it the "prodder", and it later became ubiquitous in microcontrollers, as the watchdog.

We therefore came up with a software analog, called "Prod". This is a simple independent buddy program, which is started automatically by the main program. It has two functions:

a) It sends regular messages to the main window of the main program. If this does not respond promptly, it forces the main program to close.
b) It tests regularly for the presence of the main program, and if it is not present, it restarts it, and closes itself.

The latter also deals with another issue. Sometimes, Windows simply closes a misbehaving application without calling its exception handler. This may be because the Forth program has corrupted its own code.

The Prod program satisfies the "simple and foolproof" test - it is only 150 lines long.
The important work is done by a 1s Windows timer.

```
: PRODWIN-TIMER ( hwnd,mess,wparam,lparam---res ) \ 1 second timer
  1 RESTART-TIMER +!                             \ Increment restart timer
  NULL ABS>REL PARAMETER-BUFFER $>ASCIIZ         \ See if main window still
there
  WINFINDWINDOW IF
    0 RESTART-TIMER !                            \ Clear restart timer
  THEN
  1 HUNG-TIMER +!                                \ Increment hung timer
  HTRACKNET @ ISHUNGAPPWINDOW 0= IF              \ Program is responding
    0 HUNG-TIMER !                               \ Clear hung timer
  THEN
  HUNG-TIMER @ HUNG-TIME U> IF                   \ Exceeded hung time
    HTRACKNET @ FALSE TRUE ENDTASK DROP          \ Kill it
  THEN
  RESTART-TIMER @ RESTART-TIME U> IF             \ Exceeded restart time
    START-TRACKNET                               \ Start the main program
    PRODWIN-CLOSE                                \ Close self
  ELSE                                           \ No message yet
    4DROP 0                                      \ Continue
  THEN
;
```

7. Division by zero, or overflow, and what to do about it
It always amazes me that Microsoft and Intel between them invented a problem that was never there before. Just because you divide by zero, they decide to throw a tantrum and close you down! When one had to write the code for a division by hand, this was never an issue. The correct answer to divide by zero is infinity - or at least the closest to infinity that the computer can approximate. So the fix is simply to rewrite the various Forth words to trap for zero, and give the correct answer immediately. I'm not sure, philosophically, whether there is a difference between plus and minus infinity, but I always carry the sign of the dividend into my maximum value, on the basis that I didn't really mean zero, I just meant a rather small number.

8. Race, or gridlock conditions and what to do about them

These problems are usually caused by incorrect interaction between threads. For example, thread A is waiting for X to be set before it sets Y. But thread B is waiting for Y to be set before it sets X.

Again, experience with microcontrollers is useful. These devices often have a large number of potential hardware interrupts triggered by peripherals that require attention - for example, when a serial port has received a character. It is very tempting to write an interrupt service routine for each peripheral in use. But it soon becomes clear that the interaction between the various interrupt service routines (ISRs) and the main program is a major source of programming errors. It is much safer to use a single, timer-driven ISR, and poll all the peripherals. This eliminates all inter-ISR problems. The minimum possible work should be done by the ISR itself, and its interaction with the main program should use only one integer for each distinct operation, so that the interrupt never needs to be disabled and re-enabled to prevent partial data update.

The same principles can be extended to Windows threads.

a) Keep the number of threads to the absolute minimum. For example, if several processes all need attention every 100ms, they should all be called from the same thread. Then, the grouped processes can communicate with each other without special consideration.
b) Do only the absolutely necessary work within the thread - usually either time-consuming, or time-critical operations.
c) Use the Windows messaging system for thread to window communication - it is debugged more thoroughly than anything you could write yourself, and handles all the hard bits.
d) Avoid all the locking mechanisms (such as critical sections) like the plague. They are a prime cause of software errors.
e) Instead, define extremely simple and exactly specified inter-thread communication using single integer reads and writes that cannot be interrupted by the task scheduler.

9. Conclusion

A reliable Windows program can be written in Forth by using a combination of techniques including fault tolerance, highly targeted detection, and adherance to strict programming principles.

NJN
September 2011