Forth versus the Beast
Taming an aggressive watchdog

Malcolm Bugler - Malvatronics Ltd, UK/USA

Abstract

As a protection mechanism against code runaway, the watchdog has become almost ubiquitous in modern processor controlled products. In fact, many processor chips, even of very low cost, have this circuitry integrated right into their basic design.

When using the popular languages C and C++ in embedded systems, debugging the root cause of a watchdog reset after the fact can be a daunting task, often requiring expensive and complicated logic analysis equipment together with specially crafted debugging code.

In contrast, the Forth run time environment provides many standard debugging features and can provide a unique set of analysis tools, allowing complete watchdog reset ‘post mortems’ to be performed right on the target, usually without any requirement for additional hardware or software.

This paper explores some ways the basic features of Forth can be used to cost and code efficiently provide these tools in a typical embedded system, and goes on to provide details of a real life application of these tools in a mission critical system.

Some Watchdog Basics

The Simple Watchdog

The task of a watchdog is to bring software that is behaving incorrectly back under control. In its simple form this is some type of hardware timer which activates the processor reset input if the software does not take specific action within a defined time period.

One method of implementing this is a logic gate driven by a RC network, the output of the gate changing state when its input threshold is exceeded. In order to ‘kick’ a watchdog of this type, a transistor or other device used to discharge the RC circuit under control of a processor output, thus holding off the reset. The discharge device is usually arranged to be pulse driven to prevent the RC network being held off if the output from the processor remains in the active state. See figure 1 below.

The software ‘kick’ is usually arranged to occur once every loop of the top-level software task. The period of the watchdog is usually set just slower that the longest time the software takes to complete one loop.

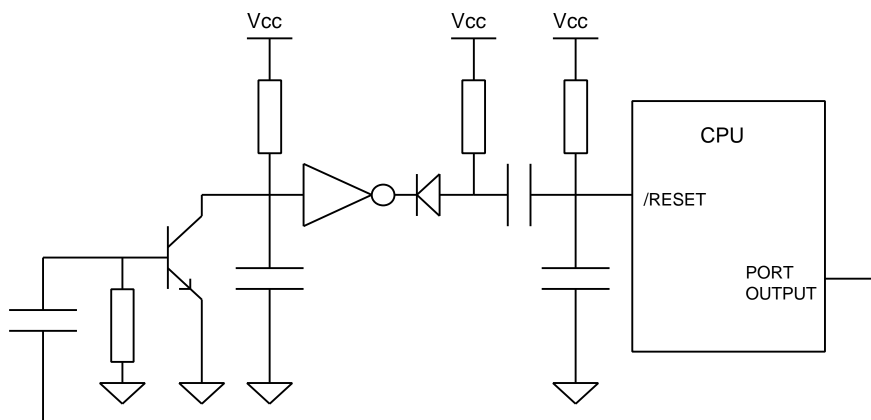

Figure 1 - Basic Watchdog Schematic

‘Better’ Watchdogs

There are many watchdog devices available currently which provide many more features than the above simple approach. Although adding a watchdog to a system is very much better than not having one, just adding the hardware is only the first step. Other things that need to be considered are:-

1) Feedback is needed by the system that watchdog activation actually caused the prior reset.
2) Any watchdog that activates the reset pin on the processor will result in loss of contextual information about where the code was immediately prior to the watchdog timeout.

Saving the Context

This means remembering where the code was immediately before a watchdog reset occurred. In order to provide this facility, an interrupt input is required rather than the CPU reset pin. This facilitates the saving of contextual information such as register contents, stacks etc, before returning the system to a ‘safe’ state. An ideal way of performing this is to use the processor Non-Maskable Interrupt or NMI input, if it has one. This type of interrupt is always active, unlike maskable interrupts, which can be disabled.

With modern watchdogs and careful design of the context saving code, it should be possible to save almost the entire system context prior to the watchdog so that a ‘post mortem’ can be performed after the event, with sufficient information to reveal the cause(s).

The Watchdog as a Beast

Problems with Conventional Environments

So, you incorporate all this into a new AC motor controller product that is undergoing on-site beta trials. Everything is going well until there is a glitch and the system reports a watchdog reset. Although there is some concern, you feel confident that with the well designed context saving watchdog code you should be able to locate and correct the problem with comparative ease. But wait; there is one issue that you may have overlooked. You have this captured information in an embedded system half way up a 40 metre tower crane, how do you get the data out and analyse it without having to hang an emulator on the system or use a run time debugger?

FORTH to the Rescue

If you had the forethought to use FORTH to code your system, there is hope after all. The very nature of a FORTH system is that it provides interactive target debugging, and most systems provide a serial port to do this through. Now all we need are some well written tools….

Post Mortem Tools

The first requirement is to provide a method to save the context in a non-volatile area of memory. This should include all processor registers, FORTH registers and the user areas as well. The watchdog should be arranged to trigger an NMI, which should stack the CPU registers, and then save data to the non-volatile area. It is preferable, if memory space allows it, to store the data in an array, allowing for several consecutive watchdog events to be captured without losing previous captured data.

The post mortem analysis tool kit should comprise of a tool to dump out the data saved by each watchdog event, plus a tool which backs up the return stack, providing a trace of where the code was when the watchdog occurred and its path there. When debugging systems at this level, it is also valuable to save flags set on entry to each interrupt and task and reset on exit from them. This information should be saved along with the other contextual data. With careful design, a of this type tool can be made quite portable across different FORTH implementations.

A Real World Example

This example demonstrates how an effectively designed watchdog system with context saving and a set of FORTH run time tools can assist is tracking down a highly allusive problem which was only occurring once every 80-100 hours of system operation.

The product in question is a medical device providing life support, and therefore is required to be extremely reliable. The product was the second generation of a previous device that used a 16 bit FORTH running on a AM186 processor. To support increased code size needed to provide new features, the system was ported across to a 32 bit FORTH running on the same platform.

The device was in production and was subject to a factory burn-in period prior to delivery to customers. This is where the problem was first discovered. Units would usually be burned in overnight and then released for delivery to customers. However, on some occasions units were left on soak over a weekend and this is where the problem was first noticed.

The problem manifested itself as a watchdog reset, which was logged by the unit. However, initially no context saving code had been installed, so the location and cause of the reset were a mystery, especially as it was extremely hard to duplicate as it occurred so infrequently.

Much testing and analysis followed using logic analysis and other tools over a period of between 2 and 3 months. These efforts simply confirmed that the problem was indeed very elusive. Finally, it was decided to change the watchdog system, which up until then had just reset the processor, first to activate a spare interrupt using a separate hardware watchdog, and finally to activate NMI using the AM186ES on-chip watchdog.

The NMI code was designed to save the entire system context in a non-volatile area of system memory, including the processor registers and FORTH stacks. Initially this was just capable of saving the context from a single watchdog, but later it was changed to be an array of several events, allowing the context of several watchdog events to be saved. The NMI routine ends in a tight loop and the external watchdog would reset the system some time later.

The code for the initial non-array register save NMI routine is detailed below. This is designed for the MPE 8632 FORTH system, but could of course be modified to run with many other systems.

```
l: nmi                    \ -- ; save all registers in aux area
    pusha                 \ push ax, cx, dx, bx, sp, bp, si and di
    push ds               \ push ds
    push es               \ push es
    mov ax, # a-lwbseg    \ set aux segment
    mov es, ax            \ in es
    mov bx, # ss-used     \ set above stack space used
    mov es: 0 [bx], sp    \ save sp
    mov ax, # a-lwbseg    \ save the stack segment data in the aux area
    mov es, ax            \
    mov di, # 0           \ start at zero
    mov si, # 0           \
    mov cx, # ss-used     \ copy length of stacks
    mov ax, # s-lwbseg    \ stack seg to copy from
    mov ds, ax            \ into data segment
    cld                   \ copy upwards
    rep movsb             \ do the move
l: doze-off               \ go to sleep and wait for external watchdog
    jr doze-off           \ note: this could also issue a hard reset
```

What this code does is simply save the registers on the data stack and save the stack segment area in the aux segment of memory provided by the MPE FORTH 6.0 cross compiler. Obviously, in a multitasking environment there would be a requirement to save all the active task areas.

The complimentary trace tool is given below in high level forth below:-

```
: .saved                                     \ -- ; dump saved context
  cr cr                                      \ throw a couple of returns
  aux0 dup ss-used + w@ dup 0=               \ saved SP = zero?
  if   2drop ." Registers not captured"      \ yes, no data captured
       cr cr                                 \
  else + ." CPU registers:- " cr             \ no, display CPU registers
       dup cs-offset + w@ ." CS " .hex
       dup ip-offset + w@ ." : IP " .hex
       dup es-offset + w@ ." : ES " .hex
       dup ds-offset + w@ ." : DS " .hex cr
       dup bx-offset + w@ ." BX " .hex
       dup dx-offset + w@ ." : DX " .hex
       dup cx-offset + w@ ." : CX " .hex
       dup ax-offset + w@ ." : AX " .hex cr
       dup di-offset + w@ ." DI " .hex
       dup si-offset + w@ ." : SI " .hex
       dup bp-offset + w@ ." : BP " .hex
       dup flags-offset + w@ ." : FL " .hex cr
       cr ." FORTH Registers:- " cr          \ display forth registers
       dup cs-offset + w@ $10000 *           \ IP is based on code seg
       over si-offset + w@ + ." IP " .hex
       dup bp-offset + w@ ss0 +              \ RP is based on stack seg
       dup rp-save ! ." : RP " .hex
       dup old-sp-offset + .lo ss0 +         \ SP is based on stack seg
       ." : SP " .hex
       dup dx-offset + w@ $10000 *           \ display TOS
       over bx-offset + w@  +
       ." : TOS " .hex cr
       old-sp-offset + normal .lo            \ work out data stack depth
       s0 @ normal .lo swap - cell/
       ." Data Stack Depth:- " . cr
       cr ." Current Nesting:- " cr          \ dump the current nesting
       r0 @ cell- cell- .lo                  \ using the return stack
       20 0                                  \ do 20 levels
       do dup aux0 +                         \ use aux segment
          @ 2- w@                            \ get ret stack data
          >name .name 3 spaces               \ display word
          cell- dup                          \ next stack position
          rp-save @ .lo <                    \ end of ret stack
          esc? or                            \ or esc pressed?
          if leave                           \ then leave early
          endif
          i @ 5 mod 0=                       \ throw cr every 5 words
          if cr endif
       loop
       drop                                  \ clean the stack
  endif                                      \
;
```

The result of the use of the above approach in the product allowed the actual problem to be isolated in the week immediately following its implementation. The actual fault was traced to the background memory test failing due to DMA from the serial port changing the contents of the memory word as it was being tested, even though interrupts were disabled! The background memory test simply went into a tight loop on failure, causing a watchdog reset and no other error indication.

Once identified, the fault was easily duplicated and corrected. The system now functions for long periods without any issue.

Conclusions

Debugging code errors, which cause watchdog resets, either by design or accident, can be extremely time and resource consuming. However, the use of a good interactive development environment and good tools can ease the many difficulties involved. The above example shows how FORTH lends itself to producing good interactive tools that run directly on the target, allowing timely debugging without the need for sophisticated equipment.