Zero-Overhead Forth Interrupts

Garth Wilson
Whittier, California

In a previous article, I wrote about completely interactive embedded-systems software development on the target itself (*FD* XVI/1). I mentioned zero-overhead high-level Forth interrupt response, saying a description of that would have to wait.

A number of good articles have been published on providing Forth interrupt response. Without invalidating the work of others, I wanted to meet the challenge of accomplishing high-level Forth interrupt response in a much simpler way that would still work for most situations in typical indirect-threaded systems. The result is described here.

One man told me recently that he always does interrupts in assembly for speed, so he wasn't very interested in high-level interrupt service. The nice thing here is that when you eliminate the overhead, the speed increases substantially.

The zero-overhead interrupt support is very simple, adds only about 100 bytes to your overall code, and can be nested as many interrupt levels deep as you wish. No additional stacks are required. It's another natural for Forth. As usual for interrupts, some assembly is required, but very little.

I call it "zero-overhead" interrupt response because when an interrupt occurs, the Forth system moves right into the interrupt-service routine just as if it were part of the normal code. It is not necessary to first save any registers or prepare to use a different stack. Here's the summary: it is as if a new word was suddenly inserted into the executing code—a word whose stack effect is (--).

To illustrate, suppose we had an interrupt service routine (word) which, for the sake of simplicity, only consisted of

```
: ISR ( -- )  1 COUNTER +!  SYSRTI  ;
```

and Forth was executing the 2 in the line

```
PRINTER  2 SPACES  BOLD_ON
```

when an interrupt was requested. The effect would be the same as if there were no interrupts and the line had said

```
PRINTER  2  INC_COUNTER  SPACES  BOLD_ON
```

where INC_COUNTER had been defined as

```
: INC_COUNTER  ( -- )  1 COUNTER +!  ;
```

like ISR above. As you can see, the main program and the interrupt can both execute without interfering with each other, even though they use the same stacks and other resources. It is not necessary to save anything before executing the ISR or restore anything afterward. The only exception is that the SYSRTI above is just an ordinary unnest (or EXIT, ;S, etc.) which also restores the ability to accept interrupts if appropriate. You may even decide to omit the SYSRTI. If you use the SYSRTI, the semicolon after it has no effect at run time.

Since servicing the interrupt does not require saving things, the interrupt service routine does not need any more stack space than other Forth words. Assuming we already had enough stack space to run Forth normally, we shouldn't have to worry about running out just because of the interrupts.

The only possible drawback with this method is that a primitive cannot be interrupted. Whatever is requesting service must wait until the current primitive is finished. This would only be a problem *if* you have primitives that take a long time to execute, *and if* those primitives are used at the times interrupt service is requested, *and if* the interrupt can't wait that long.

Otherwise, consider that it will typically take many primitives to service the interrupt, and it would be an insignificant delay to wait for one primitive in the main program to finish executing. It typically takes far less time to finish the currently executing primitive than to do all the register-saving and other setups required by other methods of high-level-language interrupt service.

The only return-from-interrupt overhead that is almost necessary with this method is that of re-enabling interrupts. If you don't need this done on a return from interrupt, the interrupt service routine can be a normal colon definition, ending with the standard unnest which is compiled by ; (semicolon), and there will be absolutely zero overhead for return from interrupt, too.

Here comes the assembly. We have to make some small changes in NEXT that basically amount to polling, and these changes slow down the Forth execution by about

Listing One-a. Original version of NEXT (no interrupt support).

```
NEXT:   LDY   #1        ; Load Y for indirect indexing. Next, load accumulator
        LDA   (IP),Y    ; with hi byte of cell pointed to by instruction pointer.
        STA   W+1       ; Store it in hi byte of word pointer.

        DEY             ; Decrement Y. Some primitives expect Y to contain 0.
        LDA   (IP),Y    ; Load accum with lo byte of cell pointed to by instruction
        STA   W         ; pointer, & store that in lo byte of word pointer.

        CLC             ; Start addition with carry flag clear.
        LDA   IP        ; Load accumulator with instruction pointer lo byte,
        ADC   #2        ; add two to it,
        STA   IP        ; and store it back where you got it.

        BCC   next1     ; If the addition above didn't cause a carry, branch around
        INC   IP+1      ; the incrementing of the hi byte. Otherwise, increment.
next1:  JMP   W-1       ; Jump to where it says "jump indirect W", so we get a
;---------------        ; doubly indirect jump.
```

one-thirtieth (in my system). If the interrupt requests come often enough, this method will run considerably faster than other methods, since you don't have to pay a big overhead penalty.

In the F83 system where I have implemented this (with an eight-bit CMOS 6502 processor), a couple of machine-language instructions added to NEXT load a byte from memory while simultaneously examining it to see whether it is zero or not. A branch is taken if appropriate. The choices are either to continue on as usual in NEXT, or to load the word pointer with the interrupt vector instead of with the contents of the address pointed to by the instruction pointer.

Some of the time taken by the extra pair of machine-language instructions is saved by the fact that we only allow two values for the byte which is fetched to see if interrupt service is necessary. These are values we would have to load into the processor's Y register anyway, even if we could somehow execute the right part of NEXT without testing.

If there is an interrupt to service, the new part of NEXT also turns off the bit in memory which records that there is interrupt service due. This takes less time than incrementing the instruction pointer, and loading the interrupt vector into the word pointer requires no indirect addressing. This means that the nest (or DOCOL, etc.) instruction in the interrupt handler actually gets executed *sooner* than the next instruction in the main code would have been executed had there been no interruption.

My original version of NEXT (before interrupt service implementation) was right out of the public-domain fig-Forth 6502 assembly source listing. The code in Listing One-a is what it looked like. (I have put all the assembly example listings here in a format used by "normal" assemblers, and commented them profusely especially for those few readers to whom 6502 assembly language is total Greek.)

After the modification, NEXT looks like the code in Listing One-b. Notice how much shorter the code is for responding to an interrupt than for continuing on with the next instruction in the main Forth code. This makes the relative interrupt response time very short. If we were to increment the instruction pointer when going to the interrupt-handling word, then the latter would be *replacing* the next Forth instruction in the main code instead of *delaying* it.

You will need a piece of machine code at the address pointed to by the machine-recognized interrupt vector location. If interrupts are enabled, this piece of code will be executed like any short machine-language interrupt service routine as soon as the hardware interrupt-request line goes true and the currently executing machine-language instruction finishes. This code only needs to put a byte in memory which can later be tested by NEXT, and disable the machine interrupt response so that the same code doesn't get executed over and over. Mine looks like Listing Two.

Next, you will need Forth words that enable and disable interrupting. These will probably have to be primitives, since most Forths won't have any words to access the µP status register. I called them IRQOK and NOIRQ. Another primitive, IRQOK?, returns my interrupt-disable flag.

A byte in RAM called irqok? (lower case) is used as a flag to record whether or not Forth interrupts are being allowed. irqok? is checked by SYSRTI, my Forth return-from-interrupt word. When a peripheral requests interrupt, setirq (in Listing Two) disables further interrupting but leaves irqok? alone.

You will usually leave interrupts disabled while the Forth interrupt service word is executing, and re-enable them when the interrupt service word finishes. SYSRTI is nothing more than unnest preceded by a few machine-language instructions to examine the content of irqok? and set or clear the processor's interrupt-disable bit accordingly. If you don't ever need to change the value of that bit immediately upon return, you can omit SYSRTI and the service word can be like any other colon defini-

Listing One-b. NEXT modified for interrupt support.

```
NEXT:    LDY   irqnot   ; Load Y with 0 if interrupt requested, otherwise 1.
         BEQ   runISR   ; Branch if interrupt requested, else continue here.
                        ; Y=1 now for indirect indexing. Load accumulator
         LDA   (IP),Y   ; with hi byte of cell pointed to by instruction pointer.
         STA   W+1      ; Store it in hi byte of word pointer.

         DEY            ; Decrement Y to 0. Some primitives will need Y to be 0.
         LDA   (IP),Y   ; Load accum with lo byte of cell pointed to by instruction
         STA   W        ; pointer, & store that in lo byte of word pointer.

         CLC            ; Start addition with carry flag clear.
         LDA   IP       ; Load accumulator with instruction pointer lo byte,
         ADC   #2       ; add two to it,
         STA   IP       ; and store it back where you got it.

         BCS   inc_hi   ; If the above addition caused a carry, branch to increment
         JMP   W-1      ; hi byte of instruction pointer. Else you're done. Done
                        ; with two JMP's because a branch not taken saves a cycle.
inc_hi:  INC   IP+1     ; Increment hi byte of instruction pointer.
         JMP   W-1      ; You're done.
;-----------------
                          ; Pick up here if interrupt was requested.
runISR:  INC   irqnot     ; Set irqnot =1, meaning no further Forth interrupt
                          ; service requested after this yet.
         LDA   FIRQVEC+1  ; Load the word pointer with the address pointed to
         STA   W+1        ; by FIRQVEC , a user variable.
         LDA   FIRQVEC    ; Load hi byte first, then lo byte. FIRQVEC is a RAM
         STA   W          ; address which holds the Forth interrupt request
                          ; vector CFA.
         JMP   W-1        ; Jump to where it says "jump indirect W", so we get a
                          ; doubly indirect jump.
;-----------------
```

Listing Two. This registers the interrupt request for NEXT.

```
irqrouting:              ; Machine-recognized interrupt vector points here.
      JMP   (MIRQVEC)    ; Jump to address pointed to by my machine-language
                         ; interrupt vector (MIRQVEC), which is initially setirq.

setirq:                ; Use to record IRQ for NEXT. Put this address in MIRQVEC.
      STZ   irqnot     ; Record that interrupt was req'ed by storing 0 in irqnot.
      STA   tempA      ; Temporarily save accumulator in tempA to put back later.
      PLA              ; Pull saved processor status byte off of µP stack,
      ORA   #04        ; set the bit corresponding to interrupt disable,
      PHA              ; and push the revised status byte back onto the stack.
      LDA   tempA      ; Restore the accumulator content.
      RTI              ; Return from interrupt. µP status gets restored modified.
;---------------
```

tion. (If you do use SYSRTI, remember that it should be followed by the semicolon to make the compiler happy.) My SYSRTI looks like the code in Listing Three.

To allow multiple-nested interrupts, an interrupt service word must re-enable interrupts (by invoking IRQOK). If you choose to do this, you might also want to push or otherwise save the content of irqok? and change it. This is so each return from interrupt leaves the interrupt-disable flag in the appropriate state. Obviously, if the flag is put back to the way it was just before the interrupt, it will always allow interrupts again. This is what SYSRTI will give you unless there was something in the interrupt service word that turned off irqok?. The purpose of irqok? is to tell SYSRTI whether or not to re-enable interrupts.

With indirect-threaded code, the average Forth primi-

tive takes about 80 clocks to execute on the eight-bit CMOS 6502 with no wait states. Since, on the average, an interrupt will hit in the middle of an executing primitive, and since NEXT is quicker at starting interrupt service than it is at normal code, the average interrupt response time will be about 90 clocks, or 9 µS at 10 MHz. This includes the time taken by the short machine-language routine pointed to by the machine interrupt request vector, MIRQVEC. Many of the slower microprocessors cannot respond this quickly even in machine language; so to do it in Forth with an eight-bit µP is excellent. 10 MHz is the fastest bus speed currently available on the 6502 from Western Design Center in Mesa, Arizona. This makes for about 125,000 Forth primitives per second. They will be introducing faster ones in the near future. There are also 16-bit versions (the 65816 and its derivatives) and WDC is developing a 32-bit version.

A Forth interrupt service routine that only looks at an asynchronous communications interface adapter (ACIA) might look like this:

```
: SYSIRQ  POLL_ACIA  DROP  SYSRTI ;
```

Since here we only have one possible source of interrupts, we can DROP the flag telling whether or not it was the ACIA that requested service. If we had several possible interrupt sources, our SYSIRQ might look like the code in Listing Four-a. Listing Four-b is an alternative that uses a support word. ?EXIT is just my word to factor out occurrences of IF EXIT THEN. Any prioritized polling of interrupt sources can be put or called between SYSIRQ and SYSRTI above.

Table One gives a summary of the changes and additions used to accomplish zero-overhead high-level Forth interrupt response. A list of requirements is first, followed by a list of enhancements.

If you have a processor with several interrupt inputs, each associated vector would put the appropriate interrupt handler address in the FIRQVEC variable.

If you have hardware that prioritizes interrupts and gives the processor a byte to read to determine the source of an interrupt without polling, it may be appropriate to have a look-up table to convert the byte into a CFA of an interrupt handler.

Hopefully it won't take too much head-scratching or meditation for this to all make sense. It really is quite simple as high-level interrupts go; and if multiple nesting doesn't make it irresistible, the elimination of overhead and separate stacks certainly should.

Listing Three. Forth return-from-interrupt.

```
CODE SYSRTI            ; Lay header & code field down.
        SEI            ; Start with interrupting disabled.
        LDA  irqok?    ; Load & test byte at addr IRQOK? to see if IRQs are ok.
        BEQ  unnest+2  ; If not ok, don't execute next (CLI) instruction.
        CLI            ; Else clear interrupt disable flag.
        BRA  unnest+2  ; Branch to body of unnest (1st adr after code field).
;----------------
```

Listing Four-a. Interrupt-handler that polls potential interrupt sources.

```
: SYSIRQ
   POLL_TIMER      NOT  IF
   POLL_ACIA       NOT  IF
   POLL_KEYBOARD   NOT  IF
   POLL_PRINTER  DROP   THEN  THEN  THEN
   SYSRTI   ;
```

Listing Four-b. Alternative with a support word.

```
: POLL  POLL_TIMER      ?EXIT
        POLL_ACIA       ?EXIT
        POLL_KEYBOARD   ?EXIT
        POLL_PRINTER     DROP   ;

: SYSIRQ       POLL     SYSRTI  ;
```

Garth Wilson began programming in Fortran and assembly in college in 1982. Three types of BASIC and Forth were among the languages he later used for data acquisition and automated test equipment. He wrote the code for a flight-following computer in assembly. Much of his early programming was on a series of TI and HP hand-held programmables, which he used to facilitate a wide range of work. A friend told him a little about Forth in 1985, but it wasn't until 1989 that he picked up Brodie's book and started getting to know Forth. As a project to learn Forth, he wrote a cross-assembler and linker program. He enjoyed the language immensely, and was delighted to see development time plunge. Programming has been a part of his job since 1986. Now he is part owner of an aircraft communications company. He can be reached by phone at 310-695-7054 or by mail at 11123 Dicky Street, Whittier, California 90606.

Table One. Summary of new code.

Necessary:

NEXT	Inner interpreter.
irqnot	RAM byte to record whether or not interrupt pending.
NOIRQ	Primitive to set µP interrupt disable bit (--).
IRQOK	Primitive to clear µP interrupt disable bit (--).
setirq	Machine-language interrupt routine that puts 0 in irqnot so NEXT knows an interrupt was requested.
SYSIRQ	Secondary for actual high-level interrupt service. No special rules except that it usually will have SYSRTI just before the semicolon (--).
COLD	(Modified, not new.) Before the first execution of NEXT, put 1 in irqnot, and make sure interrupts are disabled so you don't get into trouble before potential interrupt sources are set up. Invoke IRQOK and (optionally) set irqok? when Forth is ready to accept interrupts.

Optional:

IRQOK?	Primitive to read µP interrupt disable bit (-- f).
irqok?	RAM byte to record whether or not to restore interrupt capability upon return from interrupt.
SYSRTI	Primitive (unnest version for return from interrupt) examines irqok?.
MIRQVEC	Variable containing an address used by the machine interrupt-service routine for a jump indirect. Not needed if you only have one routine.
FIRQVEC	Variable containing the Forth interrupt vector. If you have more than one high-level interrupt service word, put the CFA of one of them here. NEXT uses it to load the word pointer from in order to service the interrupt.