Chapter 22

Zenning and the Flexible Mind

Chapter 22

Taking a Spin through What You've Learned

And so we come to the end of our journey; for now, at least. What follows is a modest bit of optimization, one which originally served to show readers of *Zen of Assembly Language* that they had learned more than just bits and pieces of knowledge; that they had also begun to learn how to apply the flexible mind—unconventional, broadly integrative thinking—to approaching high-level optimization at the algorithmic and program design levels. You, of course, need no such reassurance, having just spent 21 chapters learning about the flexible mind in many guises, but I think you'll find this example instructive nonetheless. Try to stay ahead as the level of optimization rises from instruction elimination to instruction substitution to more creative solutions that involve broader understanding and redesign. We'll start out by compacting individual instructions and bits of code, but by the end we'll come up with a solution that involves the very structure of the subroutine, with each instruction carefully integrated into a remarkably compact whole. It's a neat example of how optimization operates at many levels, some much less determininstic than others—and besides, it's just plain fun.

Enjoy!

Zenning

In Jeff Duntemann's excellent book *Borland Pascal From Square One* (Random House, 1993), there's a small assembly subroutine that's designed to be called from a Turbo

Pascal program in order to fill the screen or a system-memory screen buffer with a specified character/attribute pair in text mode. This subroutine involves only 21 instructions and works perfectly well; however, with what we know, we can compact the subroutine tremendously and speed it up a bit as well. To coin a verb, we can "Zen" this already-tight assembly code to an astonishing degree. In the process, I hope you'll get a feel for how advanced your assembly skills have become.

Jeff's original code follows as Listing 22.1 (with some text converted to lowercase in order to match the style of this book), but the comments are mine.

LISTING 22.1 L22-1.ASM

```
OnStack     struc       ;data that's stored on the stack after PUSH BP
OldBP       dw    ?     ;caller's BP
RetAddr     dw    ?     ;return address
Filler      dw    ?     ;character to fill the buffer with
Attrib      dw    ?     ;attribute to fill the buffer with
BufSize     dw    ?     ;number of character/attribute pairs to fill
BufOfs      dw    ?     ;buffer offset
BufSeg      dw    ?     ;buffer segment
EndMrk      db    ?     ;marker for the end of the stack frame
OnStack     ends
;
ClearS      proc  near
      push  bp                          ;save caller's BP
      mov   bp,sp                       ;point to stack frame
      cmp   word ptr [bp].BufSeg,0      ;skip the fill if a null
      jne   Start                       ; pointer is passed
      cmp   word ptr [bp].BufOfs,0
      je    Bye
Start: cld                        ;make STOSW count up
      mov   ax,[bp].Attrib        ;load AX with attribute parameter
      and   ax,0ff00h             ;prepare for merging with fill char
      mov   bx,[bp].Filler        ;load BX with fill char
      and   bx,0ffh               ;prepare for merging with attribute
      or    ax,bx                 ;combine attribute and fill char
      mov   bx,[bp].BufOfs        ;load DI with target buffer offset
      mov   di,bx
      mov   bx,[bp].BufSeg        ;load ES with target buffer segment
      mov   es,bx
      mov   cx,[bp].BufSize       ;load CX with buffer size
      rep   stosw                 ;fill the buffer
Bye:  mov   sp,bp                 ;restore original stack pointer
      pop   bp                    ; and caller's BP
      ret   EndMrk-RetAddr-2      ;return, clearing the parms from the stack
ClearS      endp
```

The first thing you'll notice about Listing 22.1 is that **ClearS** uses a **REP STOSW** instruction. That means that we're not going to improve performance by any great amount, no matter how clever we are. While we can eliminate some cycles, the bulk of the work in **ClearS** is done by that one repeated string instruction, and there's no way to improve on that.

Does that mean that Listing 22.1 is as good as it can be? Hardly. While the speed of **ClearS** is very good, there's another side to the optimization equation: size. The whole of **ClearS** is 52 bytes long as it stands—but, as we'll see, that size is hardly set in stone.

Where do we begin with **ClearS**? For starters, there's an instruction in there that serves no earthly purpose—**MOV SP,BP**. SP is guaranteed to be equal to BP at that point anyway, so why reload it with the same value? Removing that instruction saves us two bytes.

Well, that was certainly easy enough! We're not going to find any more totally non-functional instructions in **ClearS**, however, so let's get on to some serious optimizing. We'll look first for cases where we know of better instructions for particular tasks than those that were chosen. For example, there's no need to load any register, whether segment or general-purpose, through BX; we can eliminate two instructions by loading ES and DI directly as shown in Listing 22.2.

LISTING 22.2 L22-2.ASM

```
ClearS      proc  near
      push  bp                           ;save caller's BP
      mov   bp,sp                        ;point to stack frame
      cmp   word ptr [bp].BufSeg,0       ;skip the fill if a null
      jne   Start                        ; pointer is passed
      cmp   word ptr [bp].BufOfs,0
      je    Bye
Start: cld                               ;make STOSW count up
      mov   ax,[bp].Attrib               ;load AX with attribute parameter
      and   ax,0ff00h                    ;prepare for merging with fill char
      mov   bx,[bp].Filler               ;load BX with fill char
      and   bx,0ffh                      ;prepare for merging with attribute
      or    ax,bx                        ;combine attribute and fill char
      mov   di,[bp].BufOfs               ;load DI with target buffer offset
      mov   es,[bp].BufSeg               ;load ES with target buffer segment
      mov   cx,[bp].BufSize              ;load CX with buffer size
      rep   stosw                        ;fill the buffer
Bye:
      pop   bp                           ;restore caller's BP
      ret   EndMrk-RetAddr-2             ;return, clearing the parms from the stack
ClearS      endp
```

(The **OnStack** structure definition doesn't change in any of our examples, so I'm not going clutter up this chapter by reproducing it for each new version of **ClearS**.)

Okay, loading ES and DI directly saves another four bytes. We've squeezed a total of 6 bytes—about 11 percent—out of **ClearS**. What next?

Well, **LES** would serve better than two **MOV** instructions for loading ES and DI as shown in Listing 22.3.

LISTING 22.3 L22-3.ASM

```
ClearS      proc  near
      push  bp                           ;save caller's BP
      mov   bp,sp                        ;point to stack frame
      cmp   word ptr [bp].BufSeg,0       ;skip the fill if a null
      jne   Start                        ; pointer is passed
      cmp   word ptr [bp].BufOfs,0
      je    Bye
Start: cld                               ;make STOSW count up
      mov   ax,[bp].Attrib               ;load AX with attribute parameter
      and   ax,0ff00h                    ;prepare for merging with fill char
```

```
        mov    bx,[bp].Filler              ;load BX with fill char
        and    bx,0ffh                     ;prepare for merging with attribute
        or     ax,bx                       ;combine attribute and fill char
        les    di,dword ptr [bp].BufOfs    ;load ES:DI with target buffer
                                           ;segment:offset
        mov    cx,[bp].BufSize             ;load CX with buffer size
        rep    stosw                       ;fill the buffer
Bye:
        pop    bp                          ;restore caller's BP
        ret    EndMrk-RetAddr-2            ;return, clearing the parms from the stack
ClearS         endp
```

That's good for another three bytes. We're down to 43 bytes, and counting.

We can save 3 more bytes by clearing the low and high bytes of AX and BX, respectively, by using **SUB *reg8,reg8*** rather than ANDing 16-bit values as shown in Listing 22.4.

LISTING 22.4 L22-4.ASM

```
ClearS         proc  near
        push   bp                          ;save caller's BP
        mov    bp,sp                       ;point to stack frame
        cmp    word ptr [bp].BufSeg,0      ;skip the fill if a null
        jne    Start                       ; pointer is passed
        cmp    word ptr [bp].BufOfs,0
        je     Bye
Start: cld                                 ;make STOSW count up
        mov    ax,[bp].Attrib              ;load AX with attribute parameter
        sub    al,al                       ;prepare for merging with fill char
        mov    bx,[bp].Filler              ;load BX with fill char
        sub    bh,bh                       ;prepare for merging with attribute
        or     ax,bx                       ;combine attribute and fill char
        les    di,dword ptr [bp].BufOfs    ;load ES:DI with target buffer
                                           ;segment:offset
        mov    cx,[bp].BufSize             ;load CX with buffer size
        rep    stosw                       ;fill the buffer
Bye:
        pop    bp                          ;restore caller's BP
        ret    EndMrk-RetAddr-2            ;return, clearing the parms from the stack
ClearS         endp
```

Now we're down to 40 bytes—more than 20 percent smaller than the original code. That's pretty much it for simple instruction-substitution optimizations. Now let's look for instruction-rearrangement optimizations.

It seems strange to load a word value into AX and then throw away AL. Likewise, it seems strange to load a word value into BX and then throw away BH. However, those steps are necessary because the two modified word values are ORed into a single character/attribute word value that is then used to fill the target buffer.

Let's step back and see what this code really *does*, though. All it does in the end is load one byte addressed relative to BP into AH and another byte addressed relative to BP into AL. Heck, we can just do that directly! Presto—we've saved another 6 bytes, and turned two word-sized memory accesses into byte-sized memory accesses as well. Listing 22.5 shows the new code.

LISTING 22.5 L22-5.ASM

```
ClearS      proc  near
      push  bp                        ;save caller's BP
      mov   bp,sp                     ;point to stack frame
      cmp   word ptr [bp].BufSeg,0    ;skip the fill if a null
      jne   Start                     ; pointer is passed
      cmp   word ptr [bp].BufOfs,0
      je    Bye
Start: cld                            ;make STOSW count up
      mov   ah,byte ptr [bp].Attrib[1] ;load AH with attribute
      mov   al,byte ptr [bp].Filler   ;load AL with fill char
      les   di,dword ptr [bp].BufOfs  ;load ES:DI with target buffer segment:offset
      mov   cx,[bp].BufSize           ;load CX with buffer size
      rep   stosw                     ;fill the buffer
Bye:
      pop   bp                        ;restore caller's BP
      ret   EndMrk-RetAddr-2          ;return, clearing the parms from the stack
ClearS      endp
```

(We could get rid of yet another instruction by having the calling code pack both the attribute and the fill value into the same word, but that's not part of the specification for this particular routine.)

Another nifty instruction-rearrangement trick saves 6 more bytes. **ClearS** checks to see whether the far pointer is null (zero) at the start of the routine...then loads and uses that same far pointer later on. Let's get that pointer into registers and keep it there; that way we can check to see whether it's null with a single comparison, and can use it later without having to reload it from memory. This technique is shown in Listing 22.6.

LISTING 22.6 L22-6.ASM

```
ClearS      proc  near
      push  bp                        ;save caller's BP
      mov   bp,sp                     ;point to stack frame
      les   di,dword ptr [bp].BufOfs  ;load ES:DI with target buffer
                                      ;segment:offset
      mov   ax,es                     ;put segment where we can test it
      or    ax,di                     ;is it a null pointer?
      je    Bye                       ;yes, so we're done
Start: cld                            ;make STOSW count up
      mov   ah,byte ptr [bp].Attrib[1] ;load AH with attribute
      mov   al,byte ptr [bp].Filler   ;load AL with fill char
      mov   cx,[bp].BufSize           ;load CX with buffer size
      rep   stosw                     ;fill the buffer
Bye:
      pop   bp                        ;restore caller's BP
      ret   EndMrk-RetAddr-2          ;return, clearing the parms from the stack
ClearS      endp
```

Well. Now we're down to 28 bytes, having reduced the size of this subroutine by nearly 50 percent. Only 13 instructions remain. Realistically, how much smaller can we make this code?

About one-third smaller yet, as it turns out—but in order to do that, we must stretch our minds and use the 8088's instructions in unusual ways. Let me ask you this: What do most of the instructions in the current version of **ClearS** do?

They either load parameters from the stack frame or set up the registers so that the parameters can be accessed. Mind you, there's nothing wrong with the stack-frame-oriented instructions used in **ClearS**; those instructions access the stack frame in a highly efficient way, exactly as the designers of the 8088 intended, and just as the code generated by a high-level language would. That means that we aren't going to be able to improve the code if we don't bend the rules a bit.

Let's think...the parameters are sitting on the stack, and most of our instruction bytes are being used to read bytes off the stack with BP-based addressing...we need a more efficient way to address the stack...*the stack*...THE STACK!

Ye gods! That's easy—we can use the *stack pointer* to address the stack rather than BP. While it's true that the stack pointer can't be used for *mod-reg-rm* addressing, as BP can, it *can* be used to pop data off the stack—and **POP** is a one-byte instruction. Instructions don't get any shorter than that.

There is one detail to be taken care of before we can put our plan into action: The return address—the address of the calling code—is on top of the stack, so the parameters we want can't be reached with **POP**. That's easily solved, however—we'll just pop the return address into an unused register, then branch through that register when we're done, as we learned to do in Chapter 14. As we pop the parameters, we'll also be removing them from the stack, thereby neatly avoiding the need to discard them when it's time to return.

With that problem dealt with, Listing 22.7 shows the Zenned version of **ClearS**.

LISTING 22.7 L22-7.ASM

```
ClearS      proc  near
      pop   dx         ;get the return address
      pop   ax         ;put fill char into AL
      pop   bx         ;get the attribute
      mov   ah,bh      ;put attribute into AH
      pop   cx         ;get the buffer size
      pop   di         ;get the offset of the buffer origin
      pop   es         ;get the segment of the buffer origin
      mov   bx,es      ;put the segment where we can test it
      or    bx,di      ;null pointer?
      je    Bye        ;yes, so we're done
      cld              ;make STOSW count up
      rep   stosw      ;do the string store
Bye:
      jmp   dx         ;return to the calling code
ClearS      endp
```

At long last, we're down to the bare metal. This version of **ClearS** is just 19 bytes long. That's just 37 percent as long as the original version, *without any change whatsoever in the functionality that **ClearS** makes available to the calling code.* The code is bound to run a bit faster too, given that there are far fewer instruction bytes and fewer memory accesses.

All in all, the Zenned version of **ClearS** is a vast improvement over the original. Probably not the best possible implementation—*never say never!*—but an awfully good one.