(continued from page 12)

fastest and just about the handiest editor under ten fingers (if it isn't, just redefine your macros the instant you think of something better). Just take the compiler and all-purpose resident macro program as bonuses.

Now, the importance of speed is simply that we want to work at programming, not at running an editor. Like a fine sound system, a good editor should be transparent. When I write *while v <= maxv do*, I don't think of the keys I press, only of the statement I'm producing. It should be the same when I want to move that statement, to indent it and what follows, or simply to erase the line—or to find it in a 2,000-line file. What no one wants is to sit and look at the editor editing.

Which brings us to WordStar imprinting. I got imprinted with WordStar because my mother imprinted me with ten fingers. For the first two days, it may be easier to use an editor that has F3 for "delete word" and shift-F3 for "delete line," but after those two days, only a WordStar-style editor gives you a chance to get to the point at which you only have to think "delete line" and not notice the actual fingerwork needed to do it—because it doesn't demand that your hands leave the typing position.

Hunt-and-peck chickens may not understand this, but there is a world beyond the barnyard, you know. The WordStar commands aren't meant for the user's manual but for the user's hands. They're the natural extension of sheer speed because they remove another nontransparent interface between your mind and the program text.

So, my ultimate editor is simply the Turbo Pascal editor with a good macro program plus several-document capacity, windowing, wordwrap for comments, block-limited search-and-replace with more complex specification capacity, and files greater than 64K. But never at the price of speed or the WordStar keyboard code. Don't mistake us far-voyaging mallards for clay pigeons.

Philippe Ranger
6120 Hutchison
Montreal, Canada H2V 4C2

```
ENTRY       STX  SAVEX     ;Save [X]
            ...            ;(Body of subroutine)
            LDX  #$FF      ;Restore [X]
SAVEX       EQU  *-1       ;Data field of preceding instruction
            RTS
```

Example 1: *A fast method for saving and restoring registers*

```
TXBLOCK     STX  _TXLENGTH   ;Set block length (0=256)
            STA  _TXADDR     ;Set block address
            STY  _TXADDR+1
            LDX  #-1         ;Init timeout
            LDY  #0          ;Init block index
            STY  _TXSUM      ;Reset TXSUM accum.
TXBLOOP1    LDA  $FFFF,Y     ;First, get byte to send
_TXADDR     EQU  *-2         ;Address portion of LDA
                              instruction
TXBLOOP2    BIT  HWREADY     ;Check if Tx port is ready
            BPL  TXBLOOP3    ;(N)Check timeout
            STA  HWDATA      ;(Y)Tx the byte
            EOR  #$FF        ;Accum TXSUM
_TXSUM      EQU  *-1         ;Data portion of EOR instruction
            STA  _TXSUM      ;Update running XOR sum
            INY
            CPY  #$FF        ;End block?
_TXLENGTH   EQU  *-1         ;Data portion of CPY instruction
            BNE  TXBLOOP1    ;(N)Continue sending
            LDA  _TXSUM      ;Send the checksum
            ...
TXBLOOP3    DEX              ;Timeout expired?
            BNE  TXBLOOP2    ;(N)Continue sending
            ...              ;(Y)Return timeout error
```

Example 2: *An actual 6502 code fragment*

6502 Hacks

Dear *DDJ*,

When I read Mark Ackerman's "6502 Hacks" (February 1987), I was both surprised and delighted that, in this world of 68000s and hypercubes, there is still anyone left who would spend the time and effort to write a good article about 6502 programming.

But, before anyone's programs start crashing, let me correct Mr. Ackerman on two related points. First, the software interrupt instruction (*BRK*) uses the *IRQ* (maskable interrupt) vector, not the *NMI* (nonmaskable interrupt) vector. But to make matters even worse, the *BRK* instruction pushes its address+2 on the return stack. In order to return to the opcode immediately following the *BRK*, the return address on the stack must be dug up and decremented before returning (very messy).

Second, and more important, is that the *RTI* (return from interrupt) instruction is not functionally compatible with the sequence *PLP* (pull processor), *RTS* (return from subroutine). The *RTI* instruction pulls the processor status byte and then continues execution at the address pulled from the stack, increments the address by 1, and then continues execution at the adjusted address. This compensates for the fact that the *JSR* pushes the address of the next opcode–1.

The reason for this lies in a quirk of the 6502 opcode processing. To execute the *JSR* instruction, the processor first fetches the opcode and the low byte of the address. It then pushes the current program counter (which is now pointing to the third byte of the instruction) and, finally,

fetches the high byte of the address. So, the address that gets pushed on the stack is always one byte shy of the next instruction. In contrast, the interrupt acknowledge sequence will only happen between instructions, so the progam counter is always pointing to an opcode when the interrupt return address is pushed.

In his coverage of self-modifying code, I think Mr. Ackerman missed one very useful trick. I often save registers (because they are so very scarce) by storing their contents in the data portion of a load instruction located at the end of the routine [see Example 1, page 122]. It takes 5 bytes of code but is absolutely the fastest method of saving and restoring a register (six cycles total) and both the instruction and data storage are localized in the subroutine.

To demonstrate variations of the trick, I have included a programming fragment of an actual routine [see Example 2, page 122]. The routine transmits, over an extremely fast synchronous communication port, a block of data (from 1 to 256 bytes) followed by the exclusive-ORed sum of the block. It must also make sure that the port is not "hung" by keeping a decaying timer in the *X* register. The routine is passed the address of the block in registers *A* and *Y* and the length of the block in *X*. When the loop is cooking, it can transmit a byte every 27 cycles.

Of course, I won't tell you what product this code is running in (the labels have been changed to protect the innocent), for fear that someone wouldn't buy our system if they knew I programmed this way on a regular basis.

James Bucanek
C-Si Systems
572 W. Pima
Coolidge, AZ 85228

DDJ