14 Tricky Techniques

Some early very small minicomputers had such a restricted instruction set that it was difficult to find *one* way to accomplish some operations. The Intel 8080 microprocessor has such a relatively rich instruction set that there are, as we have seen, often many ways to accomplish a desired result.

A programmer, under pressure to produce, will usually select the most straightforward coding, even if it is not the solution that is most compact or executes fastest. But when speed or program size are important, it is necessary for you to know how to get things done using a minimum of bytes of code and cycles of CPU execution time.

Since the 8080 is stack-oriented, and most earlier machines were not, older programmers often do not know some of the tricks that are available thanks to the stack. One was mentioned in Chap. 13: letting one subroutine fall through into another. There are lots more, and we will be using some of them to implement three new subroutines. You will find these additions to CPMIO to be great labor saving devices. They should be. They are emulating statements found in higher level languages. That had better make your programming easier!

TWOCR:, a one-line subroutine

Some console messages, especially error messages and warnings, should be displayed with blank lines above and below to set them off from surrounding text. Obviously, you could call CCRLF: twice in a row before and after displaying such messages. But it is easier to add one line of code to CPMIO to implement a double line feed.

Immediately above CCRLF: in your CPMIO.ASM file, add the line:

```
TWOCR: CALL   CCRLF
```

and then your programs can call TWOCR: instead of CCRLF: when you want to double space text on the console. Since this addition is called as a subroutine but has no return of its own, it will execute a call to CCRLF:, producing a carriage return and line feed. Then the return from CCRLF: will be back to the instruction following the call. That instruction is the entry point to CCRLF: so you will get another CR and LF before the original calling program is returned to.

SPMSG: displays in-line messages

In order to display our sign-on message (Chap. 13) we set aside a message text buffer, SINON:, and had to load an index register with its start address before calling our console output message subroutine, COMSG:. There is a better way to display messages, allowing the programmer to include message texts within the flow of programs, instead of setting up separate text buffers.

In the middle of a BASIC language program, you can say:

```
PRINT 'Message text for the console'
```

and the text within the quotes will be displayed for the operator. The text thus follows the instruction PRINT and precedes the next instruction (or program statement, as it is known in BASIC). The text appears at the point in the program where it will be displayed, rather than in a remote buffer area.

Using SPMSG: you can include a similar function in assembly language programs, without having to set up message text buffers, and without using even one index register! Simply write:

```
CALL  SPMSG
DB    'Message text for the console', 0
```

and SPMSG: will output your text and return to the instruction following the zero byte terminator. This allows you to place console messages within the mainstream of your program, improving program readability and reducing program size and register usage.

Later, we will be looking in detail at how the stack operations make this possible. First let's look at the test program for this chapter and see how our new subroutines are used.

LISTING 14–1. TESTC14.LIB

```
START2: CALL    TWOCR           ; DOUBLE SPACE LINES
        CALL    SPMSG           ; PROMPT FOR TEST
        DB      'TESTING FOR YES OR NO FROM CONSOLE',CR,LF,0
        CALL    GETYN
        JNZ     START3          ; GOT A "NO"
        CALL    SPMSG           ; GOT A "YES"
        DB      LF,'YOUR ANSWER WAS "YES!"',0
        JMP     START2
START3: CALL    SPMSG
        DB      LF,'YOU SAID "NO!"',0
        JMP     START2
```

This time you will not replace all of the old test program within CPMIO.ASM. Leave the first four lines to retain the sign-on message display. Delete the next six lines, START2: through the JMP START2, and then merge in TESTC14.LIB, which you will create from the program lines in List. 14–1.

This new test program first checks out TWOCR:. It then calls our stack-oriented message routine to display the text in the line immediately following. Once again, this text must be terminated by a zero, so that SPMSG: knows when to quit and return to the next opcode.

That opcode calls our final console I/O subroutine, GETYN:.

This routine will prompt the console operator for a yes or no decision, and will return to the calling program with that decision recorded in the zero flag.

GETYN: interrogates the operator

When called, this subroutine will display the short prompt:

(Y/N)?:

and wait for the operator's response. Line editing is in effect for the operator, and so an immediate CTRL C will cause a program abort and a return to CP/M. The only other valid responses are either upper- or lower-case "Y" or "N" as the first character in the line. GETYN: uses the buffered console input subroutine for operator responses, but only examines the first character on the line returned.

A valid response therefore could be "yep" or "YEAH!" or just "y" by itself, with a CR signaling that the operator is ready to have the response accepted. If the console operator answers incorrectly, with neither "y" nor "n" nor CTRL C as the first character on the line, he will be reprompted. GETYN: will return to the calling program with the zero flag set if the answer is yes, or the zero flag not set if the answer is no.

Reading through the test program in List. 14–1 will show you how all of these subroutines are used. GETYN: is called, the response tested by the conditional **J**ump on **N**ot **Z**ero, and one of two messages is displayed showing what the program thinks the operator meant.

In real programs, the single line CALL GETYN followed by a conditional jump on zero or not zero will effect a program response to the operator's wishes. Every time a yes or no answer is needed, the programmer calls SPMSG: to ask the question, and GETYN: to receive the answer.

Now it is time for you to key in the two .LIB files in List. 14–1 and 14–2, and merge them into CPMIO.ASM. To be consistent with the listings in this book, you should merge the subroutines in right before INBUF: in your existing program. Be careful with those nested quotes in the test program!

LISTING 14–2. CH14.LIB

```
; MESSAGE POINTED TO BY STACK OUT TO CONSOLE
SPMSG:  XTHL                    ; GET "RETURN ADDRESS" TO HL
        XRA     A               ; CLEAR FLAGS AND ACCUMULATOR
        ADD     M               ; GET ONE MESSAGE CHARACTER
        INX     H               ; POINT TO NEXT
        XTHL                    ; RESTORE STACK FOR
        RZ                      ; RETURN IF DONE
        CALL    CO              ; ELSE DISPLAY CHARACTER
        JMP     SPMSG           ; AND DO ANOTHER

; GET YES OR NO FROM CONSOLE
GETYN:  CALL    SPMSG           ; PROMPT FOR INPUT
        DB      ' (Y/N)?: ',0
        CALL    CIMSG           ; GET INPUT LINE
        CALL    CCRLF           ; ECHO CARRIAGE RETURN
        LDA     INBUF+2         ; FIRST CHARACTER ONLY
        ANI     01011111B       ; CONVERT LOWER CASE TO UPPER
        CPI     'Y'             ; RETURN WITH ZERO = YES
        RZ
        CPI     'N'             ; NON-ZERO = NO
        JNZ     GETYN           ; ELSE TRY AGAIN
        CPI     0               ; RESET ZERO FLAG
        RET                     ; AND ALL DONE
```

With the new test program lines and the new subroutines merged into CPMIO.ASM, you should be able to assemble, load and run the test. Your CPMIO.PRN file should match Listings 14–3 and 14–4. Only the portions of CPMIO that have been changed are included in these listings. Did you remember to patch in the one-line TWOCR: subroutine?

LISTING 14–3. A partial listing of CPMIO, showing the new code resulting from including TESTC14.LIB

```
0100 319602     START:  LXI     SP,STAK         ; SET UP USER'S STACK
0103 CDA101     START1: CALL    CCRLF           ; START A NEW LINE
0106 217101             LXI     H,SINON         ; WITH SIGN-ON MESSAGE
0109 CDAB01             CALL    COMSG
010C CD9E01     START2: CALL    TWOCR           ; DOUBLE SPACE LINES
010F CDD401             CALL    SPMSG           ; PROMPT FOR TEST
0112 5445535449         DB      'TESTING FOR YES OR NO FROM CONSOLE',CR,LF,0
0137 CDE001             CALL    GETYN
013A C25B01             JNZ     START3          ; GOT A "NO"
013D CDD401             CALL    SPMSG           ; GOT A "YES"
0140 0A594F5552         DB      LF,'YOUR ANSWER WAS "YES!"',0
0158 C30C01             JMP     START2
015B CDD401     START3: CALL    SPMSG
015E 0A594F5520         DB      LF,'YOU SAID "NO!"',0
016E C30C01             JMP     START2

0171 5349474E2DSINON:   DB      'SIGN-ON MESSAGE',CR,LF,0
```

LISTING14–4. A partial listing of CPMIO, showing the new code resulting from including CH14.LIB.

```
                  ; MESSAGE POINTED TO BY STACK OUT TO CONSOLE
01D4 E3           SPMSG:  XTHL                      ; GET "RETURN ADDRESS" TO HL
01D5 AF                   XRA     A                 ; CLEAR FLAGS AND ACCUMULATOR
01D6 86                   ADD     M                 ; GET ONE MESSAGE CHARACTER
01D7 23                   INX     H                 ; POINT TO NEXT
01D8 E3                   XTHL                      ; RESTORE STACK FOR
01D9 C8                   RZ                        ; RETURN IF DONE
01DA CD9101               CALL    CO                ; ELSE DISPLAY CHARACTER
01DD C3D401               JMP     SPMSG             ; AND DO ANOTHER

                  ; GET YES OR NO FROM CONSOLE
01E0 CDD401       GETYN:  CALL    SPMSG             ; PROMPT FOR INPUT
01E3 2028592F4E           DB      ' (Y/N)?: ',0
01ED CDB501               CALL    CIMSG             ; GET INPUT LINE
01F0 CDA101               CALL    CCRLF             ; ECHO CARRIAGE RETURN
01F3 3A0502               LDA     INBUF+2           ; FIRST CHARACTER ONLY
01F6 E65F                 ANI     01011111B         ; CONVERT LOWER CASE TO UPPER
01F8 FE59                 CPI     'Y'               ; RETURN WITH ZERO = YES
01FA C8                   RZ
01FB FE4E                 CPI     'N'               ; NON≠ZERO = NO
01FD C2E001               JNZ     GETYN             ; ELSE TRY AGAIN
0200 FE00                 CPI     0                 ; RESET ZERO FLAG
0202 C9                   RET                       ; AND ALL DONE

0203              INBUF:  DS      83                ; LINE INPUT BUFFER

                  ; SET UP STACK SPACE
0256                      DS      64                ; 40H LOCATIONS
0296 00           STAK:   DB      0                 ; TOP OF STACK
```

How SPMSG: works

The text to be displayed by this subroutine is in a buffer, just like that for COMSG: in the previous chapter. The only difference is that this buffer is embedded within the program, immediately following the CALL SPMSG:. After the subroutine is called, we still have to fetch one character at a time, test for zero, and output the character through CO: if it is nonzero.

In using COMSG: we pointed to the text by loading the HL index with the start address of the text. Now, the start address of the text is the "opcode" of the "instruction" immediately following the CALL SPMSG:. In other words, it is the return address, or would be for a usual subroutine call.

Since return addresses are pushed onto the stack by CALL opcodes, a pointer to the beginning of the message text is sitting on top of the stack, waiting for us to load it into an index register. POP H would do this, but would of course destroy the previous contents of the HL register pair. We said we were not going to "use" any indexes. That is not literally true. The subroutine will not *change* any index register contents. Effectively the same thing, as far as the programmer is concerned.

This is accomplished by the use of the super powerful 8080 instruction XTHL. This mnemonic stands for e**X**change **T**op of stack with **HL** register. The "return address" from the stack is moved into the HL pair, at the same time that the contents of the HL pair is moved to the top of the stack. Obviously the 8080 microprocessor contains some temporary holding registers that are invisible to the programmer but that permit this bidirectional simultaneous swap.

So with one instruction that is only one byte long we have both loaded the index with the start address of the message and at the same time saved the previous contents of the index. Now we have to fetch the first character from memory. Just to teach you a new instruction or two, we do this differently this time.

The logical "exclusive or" operation can be abbreviated XOR, and it executes by comparing each bit of the contents of the accumulator with each bit of a second operand. The exclusive part of this OR implies that the resultant bit should be a one if either operand bit is a one (the same as OR) but not if both bits are one (the exclusive part).

1. Zero XOR zero is zero.
2. Zero XOR one is one.
3. One XOR zero is one.
4. One XOR one is zero.

You can compare this "truth table" with those for AND and OR given in previous chapters.

The 8080 mnemonic for this operation is XRA, since the A register is always one of the operands. XRA A says take the exclusive or of A with A. For any bit pair, since both bits are the same, the truth table above says the result will always be zero. XRA A zeros the accumulator. Well, that didn't accomplish much since we want the character that is to be displayed in the A register. If we now MOV A,M to get it there, we will still have to execute some operation through the ALU to set the zero flag so we can detect the end of message terminator. Since we have zeroed A, we can add the contents of the M register to it. Zero plus any number equals the number, which doesn't accomplish anything either. Except in a computer, where the operands are added in the ALU, and the results stored in the flag register. ADD M in this case sets the zero flag if our fetched character was the terminator.

But it wasn't. We are still pointing to the first character. We add M to A, zero is not set, and now we can output the character. We can, but if we do that right away, we will lose the zero flag bit setting because of all the stuff that would be going on downstream, and we haven't tested it yet. So instead we do things in an orderly manner. First we increment the index to point to the next character. Then we reswap HL and the top of stack. Neither of these double-precision operations affect the flag bits. Now we test zero with the conditional **R**eturn on **Z**ero. If we didn't INX the index and put it back on the stack, the RZ wouldn't have the correct return address to work on. Bomb!

From this you can follow the logic of this subroutine. It fetches and displays each text character in turn, always keeping a pointer to the next character on the top of the stack. When the fetched character is the terminator, the top of the stack contains the real return address, and a return gets us back to the calling program at the instruction following the message text. Well, we wanted to do just that. How about that?

With a dozen carefully chosen bytes of code, carefully arranged in a logical sequence, we have performed a function that relieves the programmer of the tedious and error prone operations of setting up text buffers, counting their characters, and keeping track of their addresses in memory. That was the way it used to be done. And to think that some stuffy old purists consider the 8080 to be a toy. The heck with them. We won't tell them that it is not a toy, but a powerful computing machine.

How GETYN: works

Nothing much new here for you to learn. Everything has previously been covered up to the LDA opcode. LoaD A register fetches the contents of the memory location pointed to by the address portion of this three-byte instruction. That address is, in this case, the first console input character in the input buffer. Note that we have used ASM.COM's arithmetic ability to point to INBUF: plus two, since the first two locations contain the maximum length and character count values that CP/M insists on. We skip over them by letting ASM compute the address INBUF+2.

When we have fetched that first character, we mask off both the eighth bit (ASCII uses only seven, remember) and also the bit that flags the difference between ASCII upper and lower case letters. Two birds killed with one stone. And a new representation learned. The mask byte is specified here in binary, so you can see the bit pattern more easily. This makes the operation more obvious than the hex equivalent mask of 5FH.

Once lower case letters have all been converted to upper case by that operation, we only have to test the upper case possibilities by comparing with the immediate ASCII values for "Y" or "N." If the "Y" compare was true, **C**om**P**are **I**mmediate sets the zero flag. So, if the answer was yes, we return to the calling program with zero set.

If no yes, we test again for "N." If no yes and no no, we jump back to the beginning and ask the operator to try again. Wouldn't it be nice if we could reach out of our program and slap the operator's wrist? If the operator input "N," our compare sets the zero flag. We can't return with it set, because that is the signal for a yes answer. So we execute an instruction that we know will clear the zero flag. Compare with zero does just that. Now we return to the calling program.

Assuming your programs include a lot of operator prompting that can be answered yes or no, this subroutine makes the main programs a lot shorter. A simple CALL GETYN followed by the conditional jump are all the instructions the main program has to execute. We have loaded up the subroutine with as much of the burden as possible, even returning with the decision information carried in a simple-to-test flag register bit.

This sort of thing is the "why" of subroutines. It is also the mark of a well designed program. If the main program consists of a string of subroutine calls with little intervening activity, it shows that the programmer did plan ahead, and that a library of subroutines was created to perform each of the necessary tasks. Sound familiar?

The end of I/O subroutines

With our CPMIO library all completed and checked out, we are ready to put together some subroutines that will get files from the disks and write files back to disks. Then you will be ready to do some real system programming. If you can't think of any more programs that need to be written, don't worry. There will be lots of ideas for you to work on in the final chapter. Your labors are only beginning.