The Practical Aspects Of Assembly Language Programming

Bruce D. Carbrey
Raleigh, NC

Part I: Using Flags

It starts with a vague sense of dissatisfaction with the limitations of BASIC. Then you feel a twinge of jealousy towards that mysterious cult of software Gurus who seem to use black magic to exhort their machines to run devilishly fast and speak in strange tongues to devices like PIAs and UARTS. Before you know it you're TAKING THE PLUNGE (**COMPUTE!**, March, 1981), and after struggling down a river of addressing modes and across a sea of opcodes, you know you've passed the initiation rites and can call yourself an Assembly Language Programmer. But perhaps you still feel like something of a novice programmer when it comes to assembly language. If you know assembly language, but don't feel confident that your machine language routines are the best that they can be, this series of articles may help. Even if you are an "expert" assembly language programmer, you may find a useful technique or two presented. Or, you may know of better techniques, in which case I encourage you to write them up and send them in to **COMPUTE!**, so we can all benefit.

I'm going to cover a number of loosely-related topics in detail, putting emphasis on program efficiency. After all, it is almost axiomatic that if you are programming in assembly language at all, you are doing it either to improve execution speed or to reduce program size, or both. The rest of this article assumes that you have a basic working knowledge of 6502 assembly language. The first installment discusses the deceptively simple topic of flags.

Representing Flags

Flags are familiar to any experienced programmer. A flag is a variable which can have only two possible states: TRUE or FALSE. It can be represented by a single bit in memory, but for ease of manipulation by a program, a whole byte is usually used.

Since flags are so simple in concept, you may be surprised to know that many programmers use flags quite inefficiently. To demonstrate what I mean, first consider the example program in Listing 1. This subroutine, usually called a keyboard driver, reads one character from an ASCII-encoded keyboard. The keyboard is assumed to be connected to a parallel I/O port such as is found in a 6820, 6522, 6530, 6532, or similar device. The seven data lines from the keyboard are tied to bits 0 through 6, and a negative-going strobe is connected to bit 7 of the port. When bit 7 of the port becomes zero, the ASCII code for the key which is depressed can be read on the remaining 7 bits. Notice that the strobe is connected to bit 7 because bit 7 is always zero in the ASCII code anyway, and because we can test it easily using BMI or BPL instructions, since bit 7 is the sign bit in a word.

Now suppose that you discover that your Monitor program will accept only uppercase alphabetic letters for commands, but your keyboard only delivers lower case letters unless you hold down SHIFT. What can you do about this nuisance if you don't have an ALPHA LOCK key? You

Properly used, flags can greatly simplify and improve your programming.

could go to the parts box and build a circuit to modify your keyboard, or you can take the software approach and simply add some code to your driver to "fold" all lower-case alphabetic characters ($61 through $7A in the ASCII table) to their uppercase equivalents, as shown below:

```
FOLD    CMP   #$7B    ;LOWER CASE "Z" + 1
        BCS   FOLD1   ;BRANCH IF NOT LOWER CASE ALPHA
        CMP   #$61    ;LOWER CASE A
        BCC   FOLD1   ;BRANCH IF NOT LOWER CASE ALPHA
        SBC   #$20    ;ELSE FOLD LOWER TO UPPER CASE
                      ALPHA
FOLD1   ...
```

This code can simply be inserted at the end of the keyboard driver, just before the RTS. The trouble is, your driver will now *always* return upper case alphabetic characters. This may be desirable for entering commands to the Monitor, but when you're in the Editor you may want to be able to input lower case. The solution? You need an "Alpha-Lock Enable" flag to tell the driver whether to allow lower case or not. You can start by allocating space for your flag:

```
ALFALK  .BYTE  0  ;ALPHA LOCK FLAG FOR
                  KEYBOARD DRIVER
```

Now how do you use it? The natural choice is to set

the flag to 1 if it's true and 0 if it's false. The complete driver routine using this method is shown in Listing 2.

This routine is satisfactory (because it works!), but it can be substantially improved. Notice that you had to temporarily save the returned character on the stack while you tested the Alpha-lock flag.

Next time I'll show a substantial improvement and more ways to improve efficiency.

LISTING 1: SIMPLE KEYBOARD DRIVER ROUTINE

```
                        ;
                        ;       SUBROUTINE INCH: KEYBOARD DRIVER FOR ASCII-ENCODED
                        ;       KEYBOARD WITH PARALLEL INTERFACE.
                        ;
                        ;       ADDRESSES SHOWN ARE FOR 6530 ON KIM-1 COMPUTER.
                        ;       KEYBOARD DATA LINES TO PORT A BITS 0 TO 6,
                        ;       NEGATIVE GOING STROBE TO BIT 7.
                        ;
                        ;       ON ENTRY: NO ARGUMENTS.
                        ;       ON RETURN: REGISTER A = ASCII CODE FOR KEY PRESSED;
                        ;       X AND Y PRESERVED.
                        ;
1700                    PAD     =       $1700   ;KIM PORT A DATA REGISTER ON 6530
1701                    PADD    =       $1701   ;KIM PORT A DATA DIRECTION REGISTER
                        ;
0000                            *=      $1780   ;**PROGRAM ORIGIN**
                        ;
1780 A900               INCH    LDA     #$00
1782 8D0117                     STA     PADD    ;SET PORT DIRECTION = INPUTS
1785 AD0017             INCH1   LDA     PAD     ;TEST PORT
1788 30FB                       BMI     INCH1   ;WAIT FOR STROBE PULSE
178A 2C0017             INCH2   BIT     PAD
178D 10FB                       BPL     INCH2   ;WAIT FOR END-OF-STROBE
178F 60                         RTS
                        ;
0000                            .END
NO ERROR LINES
```

LISTING 2: KEYBOARD DRIVER WITH ALPHA LOCK FLAG
USING 0 = FLASE AND NON-0 = TRUE

```
                        ;
                        ;       SUBROUTINE INCH: KEYBOARD DRIVER FOR ASCII-ENCODED
                        ;       KEYBOARD WITH PARALLEL INTERFACE.
                        ;
                        ;       ADDRESSES SHOWN ARE FOR 6530 ON KIM-1 COMPUTER.
                        ;       KEYBOARD DATA LINES TO PORT A BITS 0 TO 6,
                        ;       NEGATIVE-GOING STROBE TO BIT 7.
                        ;
                        ;       ON ENTRY: IF ALFALK IS NON-0, THEN ALL LOWERCASE LETTERS WILL
                        ;       BE RETURNED AS THE EQUIVALENT UPPERCASE ALPHA.
                        ;       ON RETURN: REGISTER A = ASCII CODE FOR KEY PRESSED;
                        ;       X AND Y PRESERVED.
                        ;
1700                    PAD     =       $1700   ;KIM PORT A DATA REGISTER ON 6530
1701                    PADD    =       $1701   ;KIM PORT A DATA DIRECTION REGISTER
                        ;
0000                            *=      $1780   ;PROGRAM ORIGIN
                        ;
1780 A900               INCH    LDA     #$00
1782 8D0117                     STA     PADD    ;SET PORT DIRECTION = INPUTS
1785 AD0017             INCH1   LDA     PAD     ;TEST PORT
1788 30FB                       BMI     INCH1   ;WAIT FOR STROBE PULSE
178A 2C0017             INCH2   BIT     PAD
178D 10FB                       BPL     INCH2   ;WAIT FOR END OF STROBE
```

Part Two:

The Practical Aspects Of Assembly Language Programming

Bruce D. Carbrey,
Raleigh, NC

Editor's Note: Last month, in the first part of this article, the author explored some methods of handling flags. At the end, he discussed setting aside bytes for flags. Here he introduces some additional techniques. To begin with, he proposes a more efficient method of storing and testing flags. RM

If instead you choose $80 to represent true and $00 to represent false, you can use the BIT instruction to test the flag without having to save the A register:

```
BIT   ALFALK  ;TEST THE FLAG
BPL   FOLD1   ;BRANCH IF NO "FOLDING" DESIRED
```

You don't have to save A because the BIT instruction sets the sign flag according to the status of bit 7 of the operand, without altering the accumulator. This saves you 4 bytes in your program, as shown in Listing 3. It also runs faster. You now know two rules to improve efficiency:

Rule 1: Use bit 7 of a byte as a flag.

Rule 2: A flag in memory can be tested without "clobbering" a register by using the BIT instruction.

Now that you know how to test the flag, you will want to be able to set or clear it. This may seem terribly obvious, for example,

```
LDA   #$80
STA   ALFALK  ;ENABLE ALPHA-LOCK MODE
```

sets the flag and,

```
LDA   #$00
STA   ALFALK  ;DISABLE ALPHA-LOCK MODE
```

clears the flag. This method uses one less byte to set the flag and two less bytes to clear the flag! On the

negative side, it takes two machine cycles longer than the first method to set the flag, but is equally fast for clearing the flag. The shift-method also does not clobber the A register, which may often be useful. Again on the negative side, you could argue that the shift method is not as straightforward as the first method, and also that it leaves the remaining seven bits of the flag "undefined". However, this can also be useful, as I shall now demonstrate.

Suppose at some point in your program you want to *temporarily* allow entry of lower case letters, and then *restore* the previous mode (either alpha-lock or non-alpha-lock, whichever was previously in effect). One method might be:

```
LDA   ALFALK  ;RECALL PRESENT ALPHA-MODE
                STATUS FLAG
PHA           ;SAVE ON STACK
LDA   #0
STA   ALFALK  ;DISABLE ALPHA LOCK TEMPORARILY
...
(code using lower case input...)
...
PLA           ;RECALL ORIGINAL ALPHA-LOCK
                STATUS
STA   ALFALK  ;RESTORE OLD MODE
```

This program segment uses the stack to save and restore the flag status. Now consider this alternative:

```
LSR   ALFALK  ;SAVE OLD MODE, CLEAR ALPHA LOCK
...
(code using lower case input...)
...
ASL   ALFALK  ;RESTORE PREVIOUS ALPHA LOCK
                MODE
...
```

This program segment performs the same function in 6 bytes instead of 13, runs faster, and doesn't clobber the accumulator! It illustrates a simple but powerful fact:

Rule 3: A single byte can be used as an 8-level push-down stack for flags.

Shifting the flag byte right moves the previous status into bit 6; shifting the flag left restores the old flag back into bit 7. This rule has several corollaries which are occasionally useful:

Rule 4: You can test the previous (saved) flag by using a BIT instruction followed by a BVC or BVS instruction.

... programs will have fewer branches, will use less memory, and will run faster ...

Rule 5: You can test both flags (bit 7 and bit 6) with only one BIT instruction.

For example:

```
BIT   FLAG    ;TEST THE FLAG
BMI   NEWSET  ;BRANCH IF PRESENT FLAG IS SET
BVS   OLDSET  ;BRANCH IF PREVIOUS FLAG WAS SET
...
```

Another side effect is:

Rule 6: You can test a flag and restore it to its previous state at the same time by using ASL followed by BCC or BCS.

For example:

```
ASL   ALFALK  ;DISCARD PRESENT, RESTORE OLD
                FLAG
BCS   ISSET   ;BRANCH IF DISCARDED FLAG WAS
                SET
...
```

The same sequence can be used to clear the flag instead if it was initialized to 0 originally and was not used as a stack. All these functions have the advantage of not disturbing any registers (except the PSW). Since they are slightly "tricky", you should document your code with clarifying comments.

As you can see, there's more to the simple little flag than meets the eye! Properly used, flags can greatly simplify and improve your programming. If you try the techniques presented here, I think you will find that your programs will have fewer branches, will use less memory, and will run faster. In next month's installment, we will look at methods for improving machine language loops.

Listing 3: Improved Keyboard Driver With Alpha-Lock Flag Using Bit 7 = 1 = True

```
;
;       SUBROUTINE INCH: KEYBOARD DRIVER FOR ASCII-ENCODED
;       KEYBOARD WITH PARALLEL INTERFACE.
;
;       ADDRESSES SHOWN ARE FOR 6530 ON KIM-1 COMPUTER.
;       KEYBOARD DATA LINES TO PORT A BITS 0 TO 6,
```

```
                ;       NEGATIVE-GOING STROBE TO BIT 7.
                ;
                ;       ON ENTRY: IF ALFALK BIT 7 IS 1, THEN LOWERCASE LETTERS WILL
                ;       BE RETURNED AS THE EQUIVALENT UPPERCASE ALPHA.
                ;       ON RETURN: REGISTER A = ASCII CODE FOR KEY PRESSED;
                ;       X AND Y PRESERVED.
                ;
1700            PAD     =       $1700       ;KIM PORT A DATA REGISTER ON 6530
1701            PADD    =       $1701       ;KIM PORT A DATA DIRECTION REGISTER
                ;
0000                    *=      $1780       ;PROGRAM ORIGIN
                ;
1780 A900       INCH    LDA     #$00
1782 8D0117             STA     PADD        ;SET PORT DIRECTION = INPUTS
1785 AD0017     INCH1   LDA     PAD         ;TEST PORT
1788 30FB               BMI     INCH1       ;WAIT FOR STROBE PULSE
178A 2C0017     INCH2   BIT     PAD
178D 10FB               BPL     INCH2       ;WAIT FOR END OF STROBE
                ;
                ;       IF ALPHA-LOCK FLAG IS SET, FOLD ANY LOWERCASE LETTERS TO
                ;       EQUIVALENT UPPERCASE LETTERS.
                ;
178F 2C9F17     FOLD    BIT     ALFALK      ;TEST "ALPHA LOCK" FLAG
1792 100A               BPL     FOLD1       ;BRANCH IF NO FOLDING DESIRED
1794 C97B               CMP     #$7B        ;LOWER CASE "Z" + 1
1796 B006               BCS     FOLD1       ;BRANCH IF PUNCTUATION
1798 C961               CMP     #$61        ;LOWER CASE "A"
179A 9002               BCC     FOLD1       ;BRANCH IF NOT LOWER CASE ALPHA
179C E920               SBC     #$20        ;ELSE FOLD TO EQUIVALENT UPPERCASE
179E 60         FOLD1   RTS
                ;
                ;       ALPHA LOCK FLAG (DEFAULT = ALLOW LOWER CASE)...
                ;
179F 00         ALFALK  .BYTE   0           ;"ALPHA LOCK" FLAG; NON-0=UPPERCASE ONLY.
                ;
0000                    .END
NO ERROR LINES                                                          ©
```