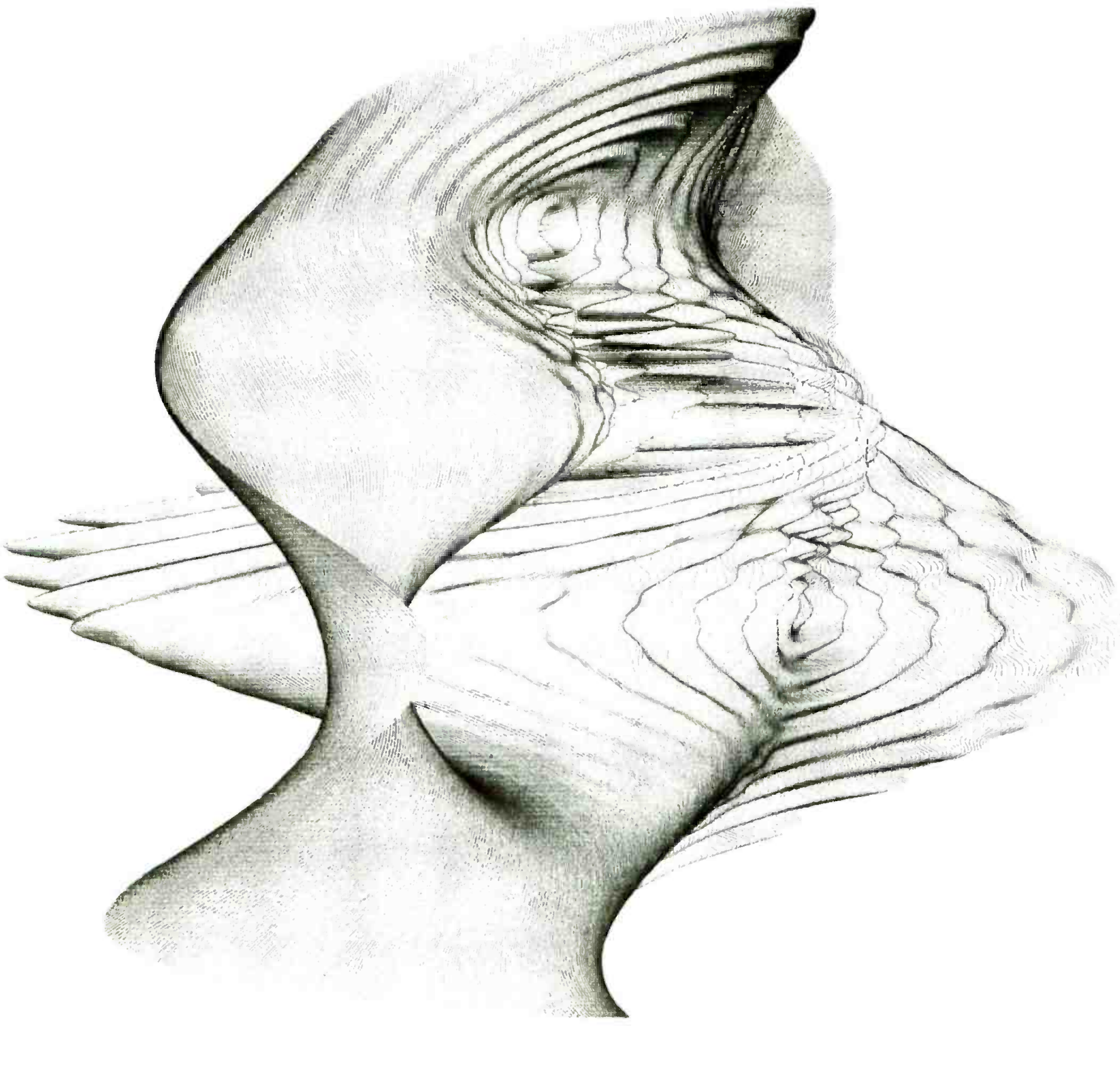
FEBRUARY 1977 VOLUME 2, Number 2 $1.50

# BYTE

the small systems journal



# A MIKBUG Roadmap . . .

John Rathkey
4808 SE 28th #316
Portland OR 97202

Some of the more useful microprocessor options for hobbyists available today are based upon Motorola's MIKBUG monitor program. These options include the Motorola 6800 Design Evaluation Kit, the SWTPC 6800 computer, etc. A big attraction of such kits is the MIKBUG read only memory, which provides the user with a monitor system and includes several utility routines. These allow the user to program in hexadecimal code from the terminal rather than in a binary code from the front panel. The purpose of this article is not to extol the virtues of the kit, but to demonstrate to proud new owners of 6800 systems that the MIKBUG read only memory can be used to much greater advantage than is generally pointed out by the manuals, particularly for IO techniques. An example of the use of MIKBUG routines is provided by the simple application of listing 1, a program which adds two numbers.

The MIKBUG firmware is a computer program for the 6800, copyright 1974 by Motorola Inc. It is called firmware because it resides in read only memory and is nonvolatile. In computers which use MIKBUG the program is located starting at hexadecimal address E000. The MIKBUG firmware takes 512 bytes, or just half of the 1 K memory. The program does not use the other half, nor does it use any device located at an address higher than hexadecimal E1FF, the end of the MIKBUG firmware. The 6800 microprocessor does use higher addresses for the interrupts and restart, but these are decoded to address locations within MIKBUG when MIKBUG is used.

The main function of MIKBUG firmware is to provide a monitor and several utility functions which make the programming and debugging processes easier. The monitor can be regarded as a home base in the vast wilderness of addressable memory. It accepts utility function commands, executes them, and returns to the terminal with an asterisk. If a program gets lost in memory, control of the situation is regained by pressing the reset button, which brings back the ever faithful servant monitor. The utility functions allow the user to load memory with a paper tape reader (L), go to any address and begin executing there (G), examine and change the contents of memory (M), print and punch selected blocks of memory (P), and display the contents of the stack, on which the values of all the registers are stored when under MIKBUG control.

To take advantage of MIKBUG one must have a terminal, and most often the beginning hobbyist will have no other peripherals to play with. Anyone who has purchased a microprocessor kit and has encountered the "let's see it do something" attitude from doubting friends and acquaintances will appreciate the immediate need for quick and easy IO techniques. Such techniques are present in MIKBUG, just waiting for the user, if he or she can find them. MIKBUG is organized in several groups of subroutines, which are selectively accessed by the MIKBUG utility functions that need them. For example, the memory change function needs a routine to input a character (M), output a character (space), input a 2 byte number (the address to be examined), input a carriage return, output a 2 byte number, space, then a 1 byte number, and so on. Many of these routines are nested several levels deep. For example, the routine to output a 2 byte number simply calls the routine to output a 1 byte number twice in a row. That's simple enough. Since a 1 byte number looks like two characters from the set zero through F to the terminal, the routine to output 1 byte uses the routine to output a character twice. As you may have guessed, input routines for numbers and characters use the same cleverness. The point of all this is that the user can use aforementioned cleverness for his or her own IO routines by simply accessing the MIKBUG subroutines at the appropriate places. People with a MIKBUG listing, familiarity with the 6800 instruction set, and the time and patience to trace through Motorola's MIKBUG mouse maze of subroutines can figure out where the appropriate places are for themselves. I encourage you to attempt this, for your own edification and purification of spirit. (It's always a good practice, when learning a new computer's instruction set, to peruse a few existing programs like MIKBUG in order to get examples.) Those lacking one of the above ingredients, or the inclination to try it, can get some of the more useful information from what I've found.

### Output Character

The output character routine, labeled OUTEEE in the MIKBUG listing, is located at hexadecimal address E1D1. It uses accumulator A as a data source. Thus you must define the contents of accumulator A which will then be interpreted as an ASCII character and shifted out in standard asynchronous format. It also uses accumulator B and the X register, but saves their contents at the beginning of the subroutine and restores them at the end. Therefore, the user need not be concerned with losing the contents of B or X. Listing 1 shows an example of the use of OUTEEE in a subroutine labeled PSTR which prints a string of characters, or a message. Control functions such as carriage return and line feed may also be implemented this way, by outputting their ASCII codes.

### Input Character

The input character routine, labeled INEEE in the MIKBUG listing, is located at hexadecimal address E1AC. Like OUTEEE it saves the X and B registers. When accessed, INEEE loops while waiting for an asynchronous format character to be sent from the terminal, and upon receiving input, shifts data into the A accumulator. After access to INEEE the content of the A accumulator is the ASCII code for the key of the terminal which was pressed when INEEE was called.

### Input Byte

This routine, labeled BYTE in the MIKBUG listing, is located at hexadecimal

*Listing 1: This example program demonstrating the uses of MIKBUG uses all the techniques discussed in the article. The program requests and inputs two 1 byte numbers. It then adds them and prints the decimal adjusted result in an algebraic sentence. The program then asks the user if another run is desired. If the reply is Y, it branches to the beginning of the program; otherwise it returns to monitor. This program requires a mere 127 bytes of memory.*

```
          M68SAM IS THE PROPERTY OF MOTOROLA SPD, INC.
             COPYRIGHT 1974 TO 1975 BY MOTOROLA INC

          MOTOROLA M6800 CROSS ASSEMBLER, RELEASE 1.2

                        NAM    ADD
0000 8D 4A     BEGIN    BSR    CARRET    CARRIAGE RETURN
0002 CE 006D            LDX    #MESS1    START ADDRESS, 1ST MESS
0005 8D 52              BSR    PSTR      PRINT MESSAGE
0007 CE 0069            LDX    #MEM      START ADDRESS, MEM BLOCK
000A BD E055            JSR    $E055     INPUT 1ST NUMBER
000D A7 00              STA A  0,X       STORE 1ST NUMBER
000F 8D 3B              BSR    CARRET    CARRIAGE RETURN
0011 BD E055            JSR    $E055     INPUT 2ND NUMBER
0014 A7 01              STA A  1,X       STORE 2ND NUMBER
0016 AB 00              ADD A  0,X       ADD NUMBERS
0018 19                 DAA              DECIMAL ADJUST
0019 A7 02              STA A  2,X       STORE ANSWER
001B 25 48              BCS    OVRFLW    BRANCH IF ANSWER 99
001D 86 20              LDA A  #$20      SPACE
001F A7 03     ANSWR    STA A  3,X       STORE 1ST ANSWER DIGIT
0021 8D 29              BSR    CARRET    CARRIAGE RETURN
0023 BD E0BF            JSR    $E0BF     OUTPUT 1ST NUMBER
0026 86 2B              LDA A  #$2B      "+"
0028 BD E1D1            JSR    $E1D1     OUTPUT "+"
002B BD E0BF            JSR    $E0BF     OUTPUT 2ND NUMBER
002E 86 3D              LDA A  #$3D      "="
0030 BD E1D1            JSR    $E1D1     OUTPUT "="
0033 A6 01              LDA A  1,X       1ST ANSWER DIGIT
0035 BD E1D1            JSR    $E1D1     OUTPUT 1ST ANSWER DIGIT
0038 BD E0BF            JSR    $E0BF     OUTPUT ANSWER
003B 8D 0F              BSR    CARRET    CARRIAGE RETURN
003D CE 0078            LDX    #MESS2    START ADDRESS, 2ND MESS
0040 8D 17              BSR    PSTR      PRINT 2ND MESSAGE
0042 BD E1AC            JSR    $E1AC     INPUT CHARACTER
0045 81 59              CMP A  #$59      =Y?
0047 27 B7              BEQ    BEGIN     IF SO REPEAT PROGRAM
0049 7E E0E3            JMP    $E0E3     OTHERWISE RETURN TO MONITOR
004C 36        CARRET   PSH A            SAVE A
004D 86 0D              LDA A  #$0D      CARRIAGE RETURN
004F BD E1D1            JSR    $E1D1     OUTPUT CARRIAGE RETURN
0052 86 0A              LDA A  #$0A      LINE FEED
0054 BD E1D1            JSR    $E1D1     OUTPUT LINE FEED
0057 32                 PUL A            RESTORE A
0058 39                 RTS              BACK TO MAIN PROGRAM
0059 A6 00     PSTR     LDA A  X         GET CHARACTER
005B 4D                 TST A            BYTE=0?
005C 27 06              BEQ    EXIT      THEN GO BACK
005E BD E1D1            JSR    $E1D1     OTHERWISE OUTPUT CHARACTER
0061 08                 INX              NEXT CHARACTER
0062 20 F5              BRA    PSTR
0064 39        EXIT     RTS              RETURN TO MAIN PROGRAM
0065 86 31     OVRFLW   LDA A  #$31      "1"
0067 20 B6              BRA    ANSWR     RETURN TO MAIN PROGRAM
0069 0004      MEM      RMB    4         RESERVE 4 SPACES
006D 32        MESS1    FCC    8,2 NO.'S?
006E 20
006F 4E
0070 4F
0071 2E
0072 27
0073 53
0074 3F
0075 0D                 FCB    $0D,$0A,0
0076 0A
0077 00
0078 4D        MESS2    FCC    6,MORE?
0079 4F
007A 52
007B 45
007C 3F
007D 20
007E 00                 FCB    0
                        END

SYMBOL TABLE

BEGIN  0000  ANSWR  001F  CARRET 004C  PSTR   0059  EXIT   0064
OVRFLW 0065  MEM    0069  MESS1  006D  MESS2  0078
```

address E055. BYTE does not affect the X register, but unlike OUTEEE and INEEE, it destroys the previous contents of the B accumulator. BYTE uses INEEE twice to get two characters, checks to be sure they are hexadecimal characters, and combines them, converting them to a 1 byte binary number in the process. This is stored in the A accumulator and is present there on return from BYTE.

### Output Byte

This routine, labeled OUT2H in the MIKBUG listing, is located at hexadecimal address E0BF. It outputs one byte of data located at some memory address chosen by the user. OUT2H requires that the X register be loaded with the address of the byte of data to be output, which may be located anywhere. This routine does not affect the contents of accumulator B, but does change the contents of accumulator A. It also increments the X register, which makes it very convenient for outputting sequentially located bytes in a block. More on this later.

*Table 1: A descriptive list of the available MIKBUG subroutines summarizes the point of this article: Don't ignore the parts and pieces of your monitor, BASIC interpreter, compiler or other programs if you intend to write assembly language or machine language code. If you buy a program, ask for a source listing so you can get programming technique pointers. (Motorola is to be commended for handing out MIKBUG listings as a standard part of documentation right from the start. MIKBUG is described in detail in Engineering Note 100, "MCM6830L7 MIKBUG/MINBUG ROM," which was published by Motorola. The program is credited to Mike Wiles and Andre Felix of Motorola Semiconductor Products Inc.)*

**Entry Points Discussed Here. . .**

| Address | Name | Description |
|---|---|---|
| E1D1 | OUTEEE | Character output: A sent to terminal device. |
| E1AC | INEEE | Character input: A set equal to next input character. |
| E055 | BYTE | Hex byte input: input two characters as hexadecimal byte in A. |
| E0BF | OUT2H | Hex byte output: A sent to terminal as two hexadecimal digits. |
| E0E3 | CONTRL | Return to MIKBUG control. |

**Other Useful MIKBUG Entry Points. . .**

| Address | Name | Description |
|---|---|---|
| E047 | BADDR | Build address by calling BYTE twice; result in X register. |
| E067 | OUTHL | Hexadecimal digit output from left nybble of byte in A. |
| E06B | OUTHR | Hexadecimal digit output from right nybble of byte in A. |
| E07B | PDATA2 | Print string of data pointed to initially by X, until EOT character (hexadecimal 04) is found. |
| E0AA | INHEX | Input hexadecimal digit, on error go to CONTRL. |
| E0C8 | OUT4HS | Output four hexadecimal characters and a space (uses OUT2H). |
| E0CA | OUT2HS | Output two hexadecimal characters and space (uses OUT2H). |

### Access to Subroutines

In the 6800 instruction set there are 16 branch instructions and two jump instructions. All may be used to access subroutines under certain conditions. The branch instructions all use relative addressing, which limits the range of branching from 126 bytes backwards to 129 bytes forwards. This is because they use a 1 byte operand as the branch offset. The jump instructions (extended addressing) use a 2 byte operand which allows them to jump anywhere. One of the branch instructions (BSR) and one of the jump instructions (JSR) store a return address in the stack before executing the branch. They go to the addresses specified by their operands and begin executing instructions at the new address until they encounter the return from subroutine instruction (RTS), at which point they return to the return addresses previously stored. Each return from subroutine instruction read by the processor must be paired with a branch or jump to subroutine instruction, although the same subroutine may be accessed by more than one branch or jump instruction. If a subroutine or a series of subroutines which is terminated with a return from subroutine instruction is accessed by any of the other branch or jump instructions, the return instruction will cause a return to an invalid address since the stack would not have been properly set up. Similarly, if a subroutine or a series of subroutines which does not end with the return instruction is accessed with jump or branch to subroutine, there will be no return to the main program. It just gets lost. The MIKBUG subroutines discussed in this article all eventually end with the return from subroutine instruction. Since they will always be located further than 129 bytes away from the main program departure point if called by a user, they must be accessed with the jump to subroutine instruction.

### General IO Techniques

More often than not, a program will need to input or output more than one character or byte at a time. The use of subroutines which access the MIKBUG subroutines facilitate this. An obvious example is the need to print a message, which involves printing several characters in a row. A good way to do this is illustrated in the subroutine labeled PSTR in listing 1. PSTR requires that the X register contain the starting address of a block of characters to be printed. PSTR increments the X register each time it prints a character and returns when it encounters an ASCII code of 00, which is a rarely used control character and is easily recognized with the test for zero (TST) instruction. Other stop characters could also be used. Similar subroutines may be used to input strings of characters or numbers. These subroutines may know when to quit by either counting the inputs and stopping at a preassigned number or by recognizing a stop character or number at the end of a string. A routine to output a string of sequentially located bytes would be even easier than PSTR using the same idea, because OUT2H increments the X register itself. Such a routine may also be terminated by either counting outputs or by recognizing a stop byte at the end of a data block. If a subroutine inputs or outputs hexadecimal numbers, it is best to count in order to terminate, otherwise one of the 256 possible numbers is excluded from use because it is the stop number. When using the decimal numbering system, any byte which is not a member of the set of 1 byte binary coded decimal numbers may be used as a stop byte.

Individual characters or small groups of characters which are input or output frequently in one program deserve their own subroutines. A good example is the combination of carriage return and line feed. The subroutine in listing 1 labeled CARRET illustrates this.

There may be times when an ouput is desired on certain conditions. There are 14 conditional branch instructions which make it easy for subroutines to serve these needs. The subroutine labeled OVRFLW in listing 1 illustrates this situation. In the sample program, if the decimal adjusted result of the addition is greater than 100, the carry bit is set and the byte reserved for the answer holds only the two least significant digits. OVRFLW is accessed if the carry bit is set, and prints a 1 in front of the answer byte to make the algebraic sentence correct.

### Return to Monitor

A happy end to any program is a graceful return to monitor. This is labeled CONTRL in the MIKBUG listing, and is located at hexadecimal address E0E3. CONTRL should be accessed with the jump (JMP) instruction, and only at the end of the users program. Listing 1 includes examples of all the routines described above. Other routines, or different nesting levels of the ones mentioned here, may be found in the MIKBUG listing, and are summarized by name in table 1. The industrious reader can find routines which may be more useful to him or her, but the preceding ones will help get the show on the terminal.■