Tinkering with Tiny BASIC

How to add four new and useful commands to Tom Pittman's brainchild, plus some tips on using USR.

Michael L. Bugg
396 Birdcage Walk
Mansfield, OH 44903

DECIMAL	HEX	USE
0032-0033	0020-0021	START OF BASIC PROGRAM (POINTER)
0034-0035	0022-0023	END OF USER MEMORY (POINTER)
0036-0037	0024-0025	END OF BASIC PROGRAM (POINTER)
0038-0039	0026-0027	TOP OF BASIC STACK (POINTER)
0040-0041	0028-0029	CURRENT LINE NUMBER
0042-0043	002A-002B	I. L. PROGRAM COUNTER
0044-0045	002C-002D	BASIC POINTER
0046-0047	002E-002F	SAVED POINTER
0048-0127	0030-007F	LINE INPUT & EXPRESSION STACK
0128-0129	0080-0081	RANDOM NUMBER SEED
0130-0181	0082-00B5	VARIABLES: 2 bytes each in order A @ 0130-0131 B @ 0132-0133 : : Z @ 0180-0181
0191	00BF	OPT COLUMN COUNTER & TAPE MODE
0256	0100	TEST FOR BREAK ROUTINE
0512	0200	COLD START-TINY BASIC
0515	0203	WARM START-TINY BASIC
0518-0520	0206-0208	JMP TO GET CHARACTER
0521-0523	0209-020B	JMP TO PRINT CHARACTER
0524-0526	020C-020E	JMP TO BREAK TEST
0532	0214	READ MEMORY BYTE SUB (PEEK)
0536	0218	STORE MEMORY BYTE SUB (POKE)
2416	0970	START OF IL CODE
2816	0B00	BASIC PROGRAM STARTS HERE-NORMAL
2897	0B51	START OF SCRATCH-PAD AREA IN MY MODIFIED TINY BASIC
3072	0C00	BASIC PROGRAM STARTS HERE-MODIFIED
6016-6118	1780-17E6	KIM: EXTRA USEABLE MEMORY
7168	1C00	KIM: START OF KIM MONITOR
7739	1E3B	KIM: PRINT HEX BYTE
7838	1E9E	KIM: PRINT SPACE
7840	1EA0	KIM: PRINT ASCII CHARACTER
7770	1E5A	KIM: INPUT ASCII CHARACTER
8093	1F9D	KIM: INPUT HEX BYTE

Table 1. Tiny BASIC decimal reference chart.

```
5    REM  ENTER HEX BYTE, PRINT DECIMAL EQUIVALENT
10   PRINT "ENTER HEX BYTE "
20   LET X = USR (8093)
30   PRINT" ";
40   PRINT X
50   END
```

Listing 1.

This article describes the USR function of Tiny BASIC and shows you how to add a few new commands to facilitate writing programs so you can replace the USR function in many instances with more understandable coding. I have also included some information and hints I found useful in tinkering with Tiny BASIC (both in modifying it and using it).

I bought a KIM-1 several years ago, but, being an avid do-it-yourselfer, I never thought I would ever buy software. I became tired of keying in programs and accidentally wiping them out by miscalculating a relative branch or missing a byte.

Tom Pittman's Tiny BASIC solved these problems. For those of us with small systems, it has to be the best software buy around. It fits quite comfortably in my 4K

```
5   REM  ENTER HEX BYTE, PRINT DECIMAL EQUIVALENT
10  PRINT "ENTER HEX BYTE ";
20  PRINT USR (8093)-0*USR(7838)
30  END
```

Listing 2.

```
5   REM  ENTER 2 HEX BYTES, PRINT DECIMAL EQUIVALENT
10  PRINT "ENTER 2 HEX BYTES ";
20  PRINT 256*USR(8093)+USR(8093)-0*USR(7838)
30  END
```

Listing 3.

memory, with room enough for my limited collection of games. (I plan on expanding the memory sometime, but for now, Tiny BASIC is it.)

Using USR

One feature of Tiny BASIC that provides unlimited versatility is the USR function. However, it was some time before I actually realized its potential. At first, I was hesitant to use it, due in part to having to calculate the addresses (and any other normally hex numbers) into decimal. However, using KIM's built-in subroutines, you can program KIM-1 (in Tiny BASIC) to perform the hex to decimal conversion for you.

The USR function is simply a machine-language subroutine call. A language such as Tiny BASIC is capable of performing almost anything you want it to do, but in some instances a machine-language subroutine is more expeditious. So, Tiny's USR is the way to break out of BASIC and execute a machine-language subroutine directly.

Listing 1 shows a simple Tiny BASIC program written for KIM using the USR function. (Other systems require adjusting the address, which this program jumps to.) This program uses one of KIM's built-in ROM monitor subroutines: the input hex byte routine (GETBYT in the KIM-1 monitor assembly listing). With this subroutine, Listing 1 converts a hex byte into its decimal equivalent.

Line 10 prints the instruction to the operator. In line 20, the variable X is made equal to whatever value is in the system accumulator upon return from the subroutine addressed by the following USR function. In this case, the value is the hex byte value obtained by packing two hex digits entered on the terminal keyboard. (Typing on the keyboard produces an ASCII-code byte for each digit entered, so this routine converts and packs them into one byte for each two entered, with the resultant byte in the accumulator register.)

When Tiny gets to this USR, it will jump to decimal address 8093, which is 1F9D in hex, the start of the GETBYT subroutine (remember: Tiny uses decimal numbers, so you will need to know the decimal equivalent of the address being jumped to). When Tiny gets here, the computer waits for the operator to punch in two hex digits on the keyboard. After the second key is accepted, the data is packed and returned to Tiny, where the variable X becomes this hex value.

CHARACTER	DECIMAL	HEX	DECIMAL X 2
A	65	41	130
B	66	42	132
C	67	43	134
D	68	44	136
E	69	45	138
F	70	46	140
G	71	47	142
H	72	48	144
I	73	49	146
J	74	4A	148
K	75	4B	150
L	76	4C	152
M	77	4D	154
N	78	4E	156
O	79	4F	158
P	80	50	160
Q	81	51	162
R	82	52	164
S	83	53	166
T	84	54	168
U	85	55	170
V	86	56	172
W	87	57	174
X	88	58	176
Y	89	59	178
Z	90	5A	180
0	48	30	
1	49	31	
2	50	32	
3	51	33	
4	52	34	
5	53	35	
6	54	36	
7	55	37	
8	56	38	
9	57	39	
+	43	2B	
-	45	2D	
/	47	2F	
*	42	2A	
.	46	2E	
RETURN	13	0D	

Table 2. Decimal equivalents.

Line 30 simply prints a space to separate the hex entry from the computer's upcoming response. A semicolon at the end of the line keeps everything on the same line. Line 40 prints the value held in variable X. Although we entered a hex value, Tiny BASIC prints its decimal equivalent. Thus, we have a program to convert from hex into decimal.

Listing 2 does exactly the same thing as Listing 1. Line 10 prints the instruction to the operator. In line 20 the PRINT command tells Tiny to print the value of the expression that follows. First, it evaluates the expression. USR (8093) comes first, so we jump to this address (just as before) to get the hex input.

The subroutine returns control back to Tiny, and the program continues. So far, the expression's value is the hex number we entered on the keyboard. The second half of the expression in line 20 starts out by subtracting zero times the value of USR (7838), which is the same as subtracting nothing. This assures that our previous value obtained will be left unchanged.

Now Tiny jumps to the subroutine at decimal 7838 (hex 1E9E). This is the system monitor's print-a-space subroutine (OUTSP in the KIM monitor listing). Keep in mind that the hex byte was already printed when we entered it through the terminal. When this second USR is executed, a space is placed just after the hex byte. Following this, we again return to Tiny, and, being at the end of the expression, the resultant value is finally printed. Since we zeroed out the second USR (assuming that the data returned in the accumulator will be useless and unknown), it has no effect on the expression's value, and our original hex number remains to be converted into decimal and printed.

This program shows what you can do with the USR function to save a little

memory space. By combining operations onto fewer program lines in this fashion, we can save that precious space in super small systems, where every byte counts.

Computing Two-Byte Addresses

Most addresses in the computer take two bytes to define, so we need to make the expression equal to a value of four hex digits entered. By modifying line 20 of Listing 2, we can create a program to convert out known hex addresses into decimal, expedite writing out those USR functions and have Tiny BASIC do our work for us.

The modification is shown in Listing 3. Note that because the subroutine called by USR (8093) only accepts one byte at a time, we must call it twice to get what we need. The first call obtains the most significant byte (MSB), so we multiply it by 256, which effectively shifts it into the proper position so Tiny evaluates it the way we want. The next call produces the least significant byte (LSB), which we add to what we already have. Finally, a call is made (as in the previous program) to print the space. The value is printed in decimal.

Using this decimal address calculator (as well as any other program using such subroutines), you must enter all four hex digits (or two for the earlier programs), including any leading zero. Also, because it is a machine-language subroutine (outside of Tiny BASIC), no input prompt is offered, and you don't have to hit the return key after entering the input. You may, of course, in a PRINT statement preceding such an input, cause a prompt of any sort to be printed.

I have used this program to work up a chart of often-used decimal addresses (Table 1). Also, a list of decimal values for some of the commonly used ASCII characters is convenient for testing data in the input buffer (Table 2). These tables, as well as this article, deal mainly with Tiny BASIC as run on a KIM-1 system starting at hex address 0200. For other addresses at which Tiny may be loaded, or other systems not having the monitor routines as listed, you would have to modify the program (but with the decimal address conversion program, this should be no problem).

Table 2 contains a column with decimal values times two. Tiny stores its variables in an address equivalent to the ASCII value of the variable name (alpha-character) doubled. For example, the location of variable A in hex is 82 (the ASCII value of A is 41, which doubled is 82) or 130 (65 times two) in decimal.

```
5     REM  HEX RELATIVE BRANCH CALCULATOR
6     REM  I= INPUT HEX SUB     S= PRINT SPACE SUB
10    I=8093
20    S=7838
30    PRINT "ENTER 'TO' THEN 'FROM' ADDRESSES (2HEX BYTES EACH) "
40    Z= USR(7739,0,256*USR(I)+USR(I)+USR(S)-256*USR(I)-USR(I)-
       USR(I)-USR(S)-2)
45    REM  USR(7739  PRINTS HEX BYTE
50    END
```

Listing 4.

Machine-Language Programming

Tiny BASIC's ability to stay together even if I make a programming mistake, along with KIM's built-in monitor subroutines, proves to be a great aid in machine-language programming. You can first program and debug complicated algorithms in BASIC and then translate them into machine language. It's easier to delete an instruction or modify the program in BASIC

than to rewrite a machine-language program to make a few changes. Once the program works properly, you can put it into machine language with Tiny helping out. Tiny BASIC can do your relative branch calculations for you. Listing 4 shows how.

Listing 4 accepts two four-digit hex addresses, automatically separates them with a space, and then prints the relative branch operand in hex. To conserve space, I used variables for the input (I) and print space (S) subroutine addresses. Table 3 summarizes the features of the USR function.

The USR functions are commonly used for two subroutines built into Tiny for reading and storing a memory byte, as PEEK and POKE in other BASICs. Although these are useful, they have one drawback: they can be difficult to follow if there are multiple USRs nested within USRs. If I review a program I had written some time ago, it takes me awhile to figure out what I had done. So, I decided to make Tiny a little bit bigger.

Adding @ and &

To make writing programs more understandable when imitating the PEEK and POKE functions of other BASICs, I modified Tiny to include a couple of new operators – @ (for one-byte numbers) and & (for two-byte numbers). Adding these to Tiny is easy.

Consider the following program line using standard Tiny BASIC syntax:

P = P – 0 * USR (538, USR (534, 46), 13)

This stores a carriage return (decimal 13) in the memory location pointed to by the line pointer (decimal 46). This is used to input string data by fooling Tiny into thinking it has come to the end of the line so that the next time an INPUT command occurs a prompt will be issued and the next input will be accepted.

Consider the following line:

LET@ @ 46 = 13

This does exactly the same thing as the previous line with the USR operation in it. This line affects no variables (normally, a USR will when used as above, so we used the "multiply by zero" trick to avoid it, such as might be necessary in a program where all variables are dedicated to something else), takes up far less program memory space and is simpler to understand at a glance.

This line uses two separate operations: the LET@ and the @ functions. These are referred to as indirection operators (from Tom Pittman's Tiny BASIC Experimenter's

FORMAT:	USR (expression)
or:	USR (expression, expression)
or:	USR (expression, expression, expression)
USE:	machine-language subroutine call, jumps to the address defined by the first expression (in *decimal*)
2nd EXPRESSION:	if included, is deposited into the processor's index registers – most significant byte goes into X-index – least significant byte goes into Y-index – (remember, all expressions become two-byte values)
3rd EXPRESSION:	if included, is deposited into the processor's accumulator register (8 bits only) – most significant byte is lost – least significant byte goes into accumulator
EVALUATION	upon return to Tiny BASIC from the machine-language subroutine the USR function becomes a two-byte value which is dependent upon the following: – Y-register value becomes most significant byte – accumulator becomes least significant byte This may be expressed as: value of USR = 256 * (X-reg) + Accum
SUMMARY:	USR (address, X and Y index registers, accumulator)
USING TINY BASIC'S BUILT-IN SUBROUTINES:	– READ BYTE (PEEK): USR(532, Address) – STORE BYTE (POKE): USR(536, Address, Data)

Table 3. USR function summary.

```
PRINT @ D
LET@ 1000 = A
LET@ A = X
LET@ A = @ X + Z
IP @ E + 40 = @ X THEN GOTO @ J
LET@ @ @ X = A * @A / @ @ 46 - USR (@C, USR (@@D, 9), @2)
```

Table 4. Using @ and LET@ operations.

✓ Reader Service index—page 241

Kit), for a poke (store) and a peek (read), respectively. This line causes the byte at the address stored at decimal 46 to equal 13. This is a form of indirect addressing.

How Indirection Works

Suppose we want to print the value of the data at address location 1000. We must enter the command

PRINT@ 1000

This prints the data at line 1000.

You may have an indirect indirection operation:

LET X = @@46

which will cause variable X to take on whatever character the input line buffer pointer (decimal address 46) is pointing to.

To alter a specified memory byte, you must add a new keyword, LET@, to Tiny. Just as before, the number following the @ sign specifies the decimal address whose byte will be set. LET@ 1000 = 0 will set address 1000 to zero.

The @ and LET@ operations can be used in most any combination (see Table 4). Since these two operations don't exist in

0285	LSB of BASIC program starting address normally 00 I left this unchanged
028C	MSB of BASIC program starting address normally 0B I changed this to 0C
097D-097E	This becomes jump to new LET& and LET@ normally 8B 4C change to 39-90
0A91-0A92	This becomes jump to new & and @ normally C1-2F change to 39-C9

Table 5. Tiny BASIC modification changes.

```
0B00  98 4C 45 54 A6     :LET& BC LET@ "LET&"   TEST FOR LET&
0B05  0A 01 22                 LN 122           YES, SET ML ADDRESS
0B08  30 BC                    JS EXPR          GET BYTE ADDRESS
0B0A  0B                       DS
0B0B  2E                       US               GO SET ADDRESS
0B0C  0C                       SP
0B0D  0A 01 29                 LN 129           SET ML ADDRESS
0B10  80 BD                    BC * "="         TEST FOR EQUAL SIGN
0B12  30 BC                    JS EXPR          GET VALUE
0B14  0B                       DS
0B15  E0                       BE *             TEST IF LINE END
0B16  2E                       US               GO DO IT
0B17  0C                       SP
0B18  1D                       NX               END OF THIS
0B19  91 4C 45 54 C0     :LET@ BC LET "LET@"    TEST FOR LET@
0B1E  0A 02 18                 LN 218           YES, SET ML ADDRESS
0B21  30 BC                    JS EXPR          GET BYTE ADDRESS
0B23  80 BD                    BC * "="         TEST FOR EQUAL SIGN
0B25  30 BC                    JS EXPR          GET VALUE
0B27  E0                       BE *             TEST IF LINE END
0B28  2E                       US               GO DO IT
0B29  0C                       SP
0B2A  1D                       NX               END OF THIS
0B2B  8B 4C 45 D4        :LET  BC BACK "LET"    TEST FOR LET
0B2F  A0                       BV *             GET VARIABLE
0B30  80 BD                    BC * "="         TEST FOR EQUAL SIGN
0B32  30 BC                    JS EXPR          GET VALUE
0B34  E0                       BE *             TEST IF LINE END
0B35  13                       SV               PUT INTO VARIABLE
0B36  1D                       NX               END OF THIS
0B37  38 19              :BACK J GOTO           BACK TO ORIGINAL CODING
0B39  C1                 :NEW  BN F40           THIS REPLACES WHAT WAS
0B3A  2F                       RT               WIPED OUT IN ORIGINAL
0B3B  89 A6              :P40  BC F41 "&"       TEST FOR &
0B3D  0A 01 15                 LN 115           YES, SET ML ADDRESS
0B40  30 BC                    JS EXPR          GET BYTE ADDRESS
0B42  0B                       DS
0B43  2E                       US               GO DO IT
0B44  2F                       RT               RETURN
0B45  89 C0              :P41  BC RET "@"       TEST FOR @
0B47  0A 02 14                 LN 214           YES, SET ML ADDRESS
0B4A  30 BC                    JS EXPR          GET BYTE ADDRESS
0B4C  0B                       DS
0B4D  2E                       US               GO DO IT
0B4E  2F                       RT               RETURN
0B4F  39 23              :RET  J  F5            GO BACK TO ORIGINAL
                                                CODING

097D  39 90 ┐
0A91  39 C9 ┘── THESE REPLACE EXISTING CODING
0B00  98 4C 45 54 A6 0A 01 22  30 BC 0B 2E 0C 0A 01 29
0B10  80 BD 30 BC 0B E0 2E 0C  1D 91 4C 45 54 C0 0A 02
0B20  18 30 BC 80 BD 30 BC E0  2E 0C 1D 8B 4C 45 D4 A0
0B30  00 BD 30 BC E0 13 1D 38  19 C1 2F 89 A6 0A 01 15
0B40  30 BC 0B 2E 2F 89 C0 0A  02 14 30 BC 0B 2E 2F 39
0B50  23
```

Listing 5.

Tiny BASIC, they must be added to it.

There are two ways of accomplishing this. The first, and best, way is to insert the coding for them into the existing interpreter at the proper points and move the following coding down with the jump addresses and adjust them accordingly. The way I do it is to blot out a part of the existing program with a jump to the new routines (which are tacked on at the end of Tiny BASIC) and have them jump back to pick up where the original coding left off. This takes a couple more bytes, but it sure beats recalculating all those jumps.

Tiny BASIC is part machine language and part intermediate language (a kind of macro-instruction programming). The modifications take place in the intermediate language (IL).

To help Tiny run faster I expanded it to include a two-byte indirection operator. It works just like the @ and LET@, except it gets and puts two bytes at a time. I use the & sign to indicate this function. This makes manipulating large amounts of data perform faster and simplifies handling variables and other values (all are two bytes). If a program had LET@ X = A and variable A was greater than 255, then part of that value would be lost (you just can't store a 16-bit value in an eight-bit byte). For timing comparisons, see Table 6.

How & and LET & Work

Suppose Z = 1. Each variable of Tiny is a two-byte value. So, in Z, the MSB is zero and the LSB is one. LET& 50 = Z will make the combined bytes 50 and 51 equal to Z. Thus, the MSB (0) will be deposited into location 51, and the LSB (1) will be put into location 50 (Tiny BASIC uses them backwards, just like the addresses in the 6502 machine-language operands: LSB comes first, then MSB).

Besides variable handling, BASIC program line numbers could be altered this way. Tables and arrays are a natural for this type of function.

To get my Tiny BASIC to learn these new things, I put the new coding at hex address 0B00 and beyond. This is where the BASIC program is normally put, so I changed the portion of Tiny that determines where the BASIC program starts. It can start just after the last byte of new coding, but I prefer to have it start at the beginning of the next

```
0115  86 C3    GET   STX$C3        STORE MSB ADDRESS  ($C2 = 0)
0117  B1 C2          LDA ($C2),Y   GET BYTE-1 (LSB ADDRESS = Y)
0119  48              PHA
011A  C8              INY
011B  B1 C2           LDA ($C2),Y   GET BYTE-2
011D  AA              TAX
011E  68              PLA
011F  A8              TAY
0120  8A              TXA
0121  60              RTS
0122  86 C3    PUT1  STX$C3        SAVE ADDRESS MSB
0124  85 E2           STA$E2                LSB
0126  60              RTS
0127  EA EA           NOP NOP
0129  A4 E2    PUT2  LDY$E2        SET INDEX
012B  48              PHA           SAVE BYTE-2
012C  8A              TXA           LOAD BYTE-1 INTO ACCUM
012D  91 C2           STA ($C2),Y   PUT BYTE -1
012F  68              PLA
0130  C8              INY
0131  91 C2           STA ($C2),Y   PUT BYTE-2
0133  60              RTS


0115                  86 C3 B1  C2 48 CA B1 C2 AA 68 A8
0120  8A 60 86 C3 85 E2 60 EA  EA A4 E2 48 8A 91 C2 68
0130  C8 91 C2 60
```

Listing 6. Source listing for machine-language coding.

USR	@	&	USR and LET
90 M=0	M=0	M=0	LET M=0
100 N=0	N=0	N=0	LET N=0
110 P=USR(536,N,0)	LET@N=0	LET&N=0	LET P=USR(536,N,0)
120 N=N+1	N=N+1	N=N+2	LET N=N+1
130 IF N<20 GOTO 110	IF N<20 GOTO 110	IF N<20 GOTO 110	IF N<20 GOTO 110
140 M=M+1	M=M+1	M=M+1	M=M+1
150 IF M<20 GOTO 100	IF M<20 GOTO 100	IF M <20 GOTO 100	IF M<20 GOTO 100
160 PRINT "END"	PRINT "END"	PRINT "END"	PRINT "END"
170 END	END	END	END
TIME = 23 SECONDS	TIME = 18 SECONDS	TIME = 9 SECONDS	TIME = 21 SECONDS

The above four programs all perform the same duties in their own way. This serves to demonstrate how programs may be rewritten to speed things up in different ways. If a program has need to move large blocks of data (such as character strings) the LET& operation can obviously speed things up considerably.

Table 6. Timing comparison tests.

page of memory (0C00 in my system) to allow room for array storage or extra variables without interference between them and the BASIC program. (This eliminates the chances of strange things happening when a program overwrites itself.)

If you are wondering why I put the new coding starting where I did, rather than directly after the existing program (originally ending at hex 0AC6). I put a multiple-statements-per-line modification (see 6502 USER NOTES, no. 13) in this gap. After a little work, Tiny BASIC doesn't act quite so tiny!

If you want to start Tiny loading the BASIC program farther down to allow room for its new growth, you can alter 0285 and 028C (this will avoid the need to enter through the warm start and set the parameters each time you start out). Hex address 028C holds the memory page number. I set this to 0C, as opposed to 0B in the original. Address 0285 holds the LSB of the starting address (normally 0). I left this unaltered.

Listing 5 contains the new coding for all of these new operations (for the IL coding), and Listing 6 shows the additional machine coding needed to accommodate it. Finally, Table 5 shows the necessary patches to the existing coding. Again, these addresses are for BASIC starting at 0200. For other starting addresses you will need to determine the changes. After you have made this modification, refer to Table 7 to remind you how to use the new operations.

Uncluttering Your CRT

Along with printing the input prompts (: and ?) and preceding a LIST operation, Tiny BASIC outputs control codes (X-on and X-off). If your system has a CRT for readout and thus has no need for these control codes, you can replace these control codes with screen control codes to make the display more readable (without the need for extra output routines to take care of business).

My TVT doesn't scroll up as it fills the screen, so after the cursor reaches the end of the page, the following output causes the cursor to wrap around to the top again, writing over what was previously there. Sometimes, this becomes quite confusing when one line ends in the middle, leaving the remains of an old line after it. Because of this, I replaced some of Tiny's control codes with the desired screen control functions: clear line, clear screen and cursor home.

@	ONE-BYTE FETCH (PEEK) – whose value is the byte at the decimal address following the @ symbol
&	TWO-BYTE FETCH (PEEK)—whose value is the combination of the two bytes at the decimal address following the & sign (LSB) and at one plus that address (MSB)
LET@	ONE-BYTE STORE (POKE)– stores a byte at the decimal address following the @ symbol
LET&	TWO-BYTE STORE (POKE) – stores a two-byte value at the decimal address following the & sign (LSB) and at one plus that address (MSB)

The addresses specified in the above operations may be any valid expression accepted by Tiny BASIC, including other similar operations, USR functions, etc.

Table 7. @, LET@, & and LET& operations summary.

At 0972 hex Tiny issues X-on after printing the colon prompt. Replace this with your choice of line or screen clear. (I use line clear.) If screen clear is used, when Tiny gives me an error code and the CRT is at the bottom line, the following colon and control code would be printed on the top line, thereby wiping out the error code before it can be read.

When inserting the code, you must alter it: set the highest bit to one. Thus, if your desired control code is 06, it must be set to 86 to insert Tiny.

The control code following the INPUT prompt (?) is located at 09DD hex. Again, observe the above instructions on setting the high bit to one.

At addresses 0A03-0A06 hex are four bytes that are printed preceding a LIST operation. These are normally all zeros, but I first insert a cursor-home control, followed by a clear-screen character. This way, the LIST starts automatically at the top of the screen and clears any previous clutter.

Also, within a program, a simple LIST Z command will clear the entire screen and put the cursor at the home position, with Z being equal to any number greater than the highest line number currently in memory. This causes nothing to be listed, so this bit of housekeeping clears the way for a clear screen so that any following output will be uncluttered. At these addresses, do not set the high bit to one as the previous ones were; simply load them as is.

Using Tiny BASIC

To squeeze long programs into small memory areas:

- Use no spaces in the programs. The programs will be difficult to read, but you will save a byte of memory for each space you don't use.
- Use abbreviations; for example, PR for PRINT or variable character for an often-used large number.
- Eliminate inessential words, for example, LET, THEN.

To speed up Tiny BASIC:

- Use variables, which are interpreted faster than numerals.
- Use the word LET. (You must decide whether speed or memory space is more important.)
- Put often-used routines into low memory. Give them the lowest line numbers.

These ideas should help you develop your own techniques to make your programs shorter and easier to write. ■