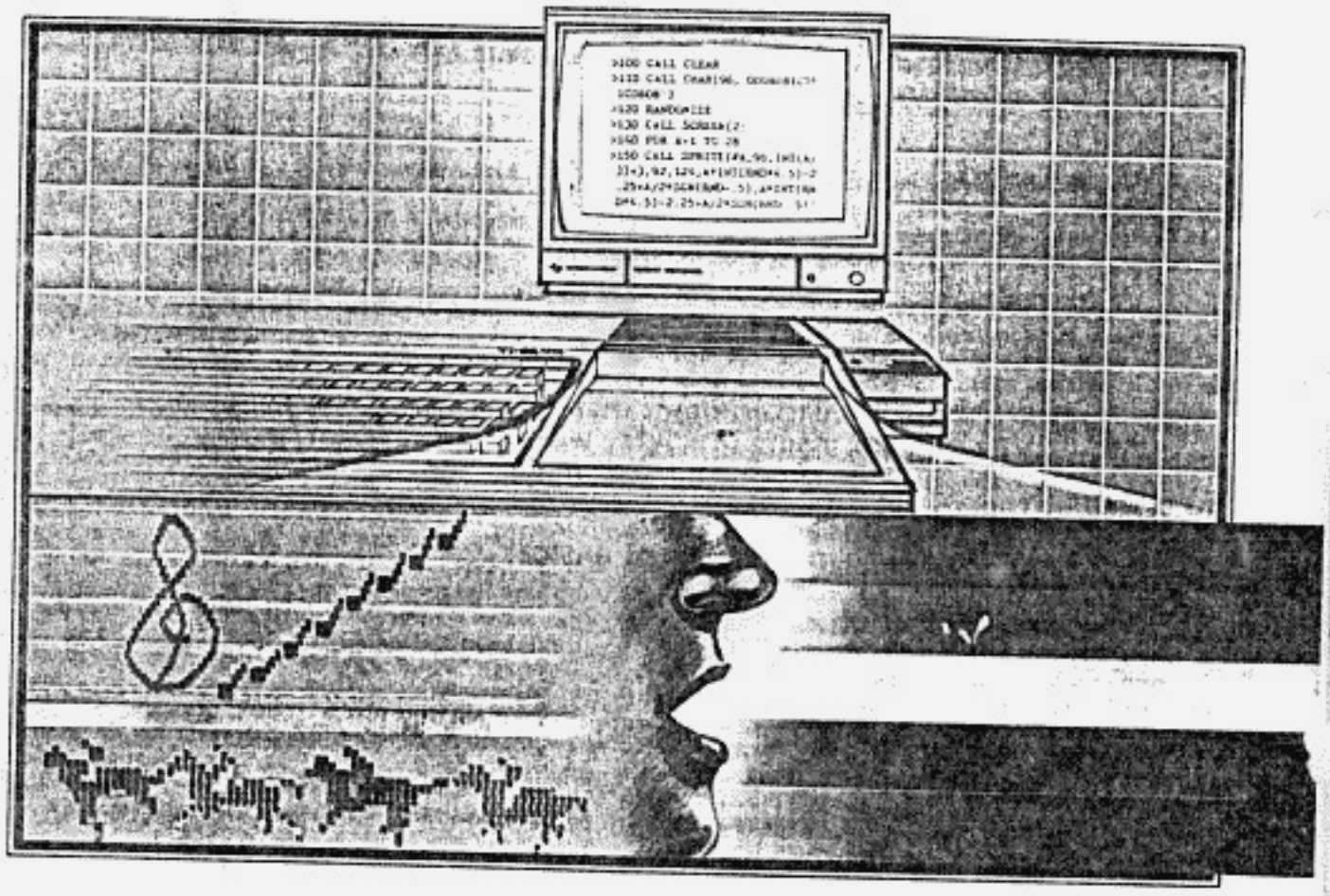
3
Inside BASIC and Extended BASIC



3

Inside BASIC and Extended BASIC

Ready to try it on your own? All it takes is BASIC logic—and a few tricks.

Language Conversion:

TRS-80 BASIC to TI BASIC

Tucked away in my basement, I have both a Radio Shack TRS-80 and a Texas Instruments TI-99/4A. The half-dozen personal computer magazines I read each month provide coding and ideas for many new programs for my TRS-80. I now have a large collection of these programs and have grown to appreciate greatly the help and enjoyment this software library provides. Unfortunately, it just hasn't been that easy to acquire software for the TI machine. [But now, with the birth of *99'er Magazine*, this situation will be rapidly remedied.—Ed.] The solution for me was obvious. I'd convert my TRS-80 programs to TI BASIC.

At the suggestion of *99'er Magazine's* editor, I read an article by Harley M. Templeton appearing in the November 1980 issue of *Personal Computing* magazine. Although the article highlighted the major differences between the versions of BASIC used on the two systems, it didn't point out which differences matter and which are merely interesting but of little practical importance. As you might expect, the only way to find out is actually to convert a program and learn from the problems that you encounter.

To set up a fair test, I selected TRS-80 programs from opposite ends of the spectrum: The first was a "number cruncher" which I had written to convert the number correct on a test to a scaled value on a continuum of learning. (My nine-to-five job involves the management of the standardized testing programs for the Portland, Oregon, School District.) The other program was an adaptation of the ideas behind a slot machine in David Ahl's *Basic Computer Games*—a program with extensive use of graphics.

The first trouble I encountered was in converting the PRINT AT command available on the TRS-80. The procedure suggested by Templeton was to set a loop as follows:

```
400 AS=" PRINT THIS STARTING AT 10,2"
500 CALL CLEAR
600 FOR I=1 TO LEN(AS)
700 N1=ASC(SEG$(AS,I,1))
800 CALL HCHAR(10,(I+1),N1)
900 NEXT I
```

In theory this works fine, but it is slow if the string length is long; single characters don't walk across the screen—they crawl! Since the program requires a prompt printed in the middle of the screen to cue the operator to enter the next five values for the scaling procedure, my final solution was to use the following coding:

```
100 PRINT "MESSAGE AT THE MIDDLE OF THE
    SCREEN"
200 PRINT : : : : : : : : : :
```

This procedure causes the text prompt to scroll up from the bottom to the middle of the screen. It is not especially speedy, but it is fast enough for the data entry in cases where you don't need lines that disappear at the top of the screen as the result of this scrolling action.

The ease with which the "number crunching" code converted was a pleasant surprise. It was important to keep track of the differences in the line numbers for GOTO's and other branches, but that, in fact, presented little problem. What was more difficult was converting the logic of IF-THEN-ELSE clauses. TRS-80 (Microsoft) BASIC allows multiple statements following the THEN- and ELSE-coding that are difficult to keep straight and re-code. The multiple line conditionals can be converted, but the conversion requires a clear head and a basic understanding of how the program works.

Because I had written the TRS-80 program myself (it had more lines of documentation than coding) and naturally understood its operation, the conversion was fairly straightforward. After I changed nearly all the PRINT and PRINT AT statements, the program worked the first time (surprise). To check it out, I made a comparison run on the TI-99/4 and the TRS-80. Surprisingly, they ran the same job in almost the same time: three minutes for a forty item test. Finally I spruced up the program a little with CLEAR and CALL SCREEN commands to take advantage of the color options available on the TI machine.

The second program was a challenge. It had essentially four main parts: (1) an introductory message, (2) the set-up graphics of the "slot machine," (3) the rotation of the wheels in the slot machine, and (4) the determinaton of the winnings and losses. The first and easiest part of the program to set up was the section which printed the introductory messages. I couldn't resist adding the CALL SCREEN command and sprucing up the comments to make it more attractive (at least to me). In this instance, the lack of speed for the HCHAR command was a benefit since it painted the screen at a leisurely-yet-pleasing pace. Before I was through, I had changed all the code in this section for aesthetic reasons.

My real conversion problems began in the second section. There, I came face-to-face with the significant dif-

ferences in the way graphics are handled by the two systems. In moving from a screen of 16×64 to one of 24×28, I had to stop and develop a new outline shape for the slot machine—one that would fit the TI screen. Deciding the colors to be used in defining the outline of the machine and the shapes to be matched (cherry, bar, bell, orange, lemon) took extra time. After some experimentation using dark blue against a white background, the lemon became a lime (dark green). To develop a new set of four characters for the orange, I experimented with CALL CHAR until the figure finally looked like a circle instead of one of Dali's exploded watches. Since there isn't an orange color available, the orange became a plum (magenta). I was still a character short, so I used the heart from the back of the user's manual.

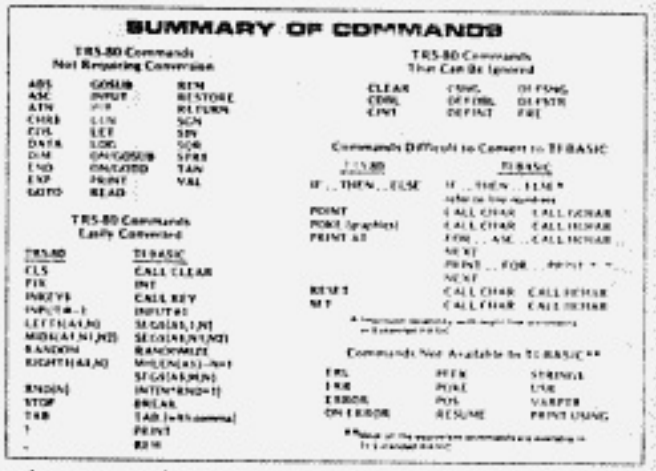
En route to coding this part of the program, I had to devise the shapes, assigning them to one of the sixteen character sets. Twice, however, I made the mistake of trying to conserve memory by using one of the character sets with pre-defined codes. This caused errors in the print statements using these codes. The moral of that experience: Whenever possible, stay away from the first eight character sets when defining new characters. It took a while to work the kinks out of this section, but the addition of color made a tremendous diffence, and I became hooked on TI graphics. (I'll probably never turn the TRS-80 version of this program again).

At this point, I realized that virtually every line of the original program had been rewritten in the move to the TI machine. Since this was to be an article on program conversion, not programming, I called the editor at *99'er Magazine* to make sure I hadn't missed the point of the article. Gary, however, wasn't surprised at all, and encouraged me to include suggestions on rewriting as well as conversion.

The third section of this program was probably the toughest to convert. I have been responsible for programming and systems analysis for over ten years on a variety of large computer systems. This has required establishing structured programming standards for every program with which I work. Even though I had personally keyed in the slot machine program, I had forgotten how poorly it was documented. This is not a criticism of Ahl's book, but rather a realistic comment on what you are likely to encounter

An Example of Code Translation From TRS-80 BASIC to TI BASIC

```
120 FOR I1=1 TO NI
130 IF IZ(I1)<>0 THEN PRINT "THIS ITEM
DROPPED";ID$: GOTO 160
140 IF K$=K1$ THEN IF C1(I1)=0 GOTO 160
ELSE C2=10*C1(I1)+200
150 DX=C3-C2
160 NEXT I1
Translates to:
120 FOR I1=1 TO NI
130 IF IZ(I1)=0 THEN 140
132 PRINT "THIS ITEM DROPPED";ID$
134 GOTO 160
140 IF K$<>K1$ THEN 150.
142 IF C1(I1)=0 THEN 160
144 C2=10*C1(I1)+200
150 DX=C3-C2
160 NEXT I1
```



SUMMARY OF COMMANDS

TRS-80 Commands Not Requiring Conversion

ABS	GOSUB	REM
ASC	INPUT	RESTORE
ATN	[illegible]	RETURN
CHR$	LEN	SGN
COS	LET	SIN
DATA	LOG	SQR
DIM	ON GOSUB	STR$
END	ON GOTO	TAN
EXP	PRINT	VAL
GOTO	READ	

TRS-80 Commands That Can Be Ignored

CLEAR	CMD	OPTION
CSAVE	DEFDBL	OUTSTR
CINT	DEFINT	PRI

TRS-80 Commands Easily Converted

TRS-80	TI BASIC
CLS	CALL CLEAR
FIX	INT
INKEY$	CALL KEY
INPUT#-1	INPUT#1
LEFT$(A$,N)	SEG$(A$,1,N)
MID$(A$,N1,N2)	SEG$(A$,N1,N2)
RANDOM	RANDOMIZE
RIGHT$(A$,N)	SEG$(A$,LEN(A$)-N+1,N)
RND(N)	INT(N*RND+1)
STOP	BREAK
TAB	TAB(-with comma)
?	PRINT
'	REM

Commands Difficult to Convert to TI BASIC

TRS-80	TI BASIC
IF...THEN...ELSE	IF...THEN...ELSE (often by line numbers)
POINT	CALL GCHAR
POKE (graphics)	CALL CHAR
PRINT AT	FOR...NEXT...CALL HCHAR
	NEXT
	PRINT...FOR...PRINT...NEXT
RESET	CALL GCHAR
SET	CALL CHAR

Commands Not Available in TI BASIC**

ERL	PEEK	STRING$
ERR	POKE	USR
ERROR	POS	VARPTR
ON ERROR	RESUME	PRINT USING

when converting a program. After an hour of tracing through a maze of GOSUBs without the benefit of a single comment, I decided on a total rewrite.

The TRS-80 version had the program determine the coordinates of one of the nine open spots on the slot machine and then perform a PRINT AT at the location. Using FOR-NEXT loops, I was able to overprint the nine spots to give the illusion of a rotating machine wheel. By converting the PRINT AT commands to HCHAR calls and storing the four codes for each shape in an array, I simulated this action on the TI-99/4. The graphics were fantastic (an unbiased estimate), but the speed was disappointing. In the TRS-80 version it was necessary to insert dummy FOR-NEXT loops to slow down the rotation of the wheels; the TI version, on the other hand, was too slow right from the start.

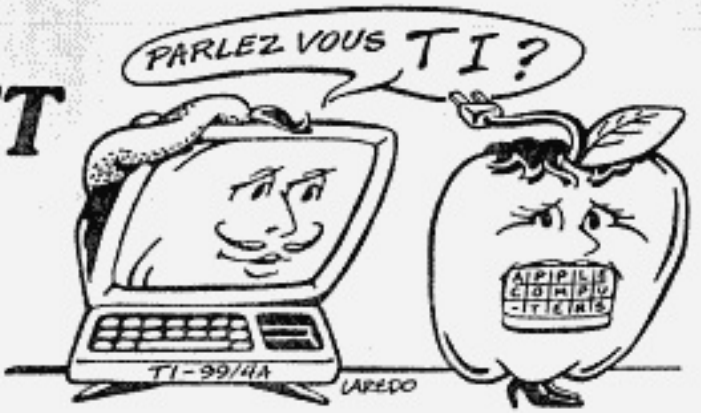
The single enhancement I had made to the TRS-80 version was to have the wheels stop one at a time, to prevent giving away the final result of the pull during rotation. To keep the wheels moving at a constant speed on the TI-99/4, I included dummy counting loops as each wheel was stopped. In spite of its lack of speed, the richness of the TI-99/4 graphics made the TI BASIC program a more appealing simulation of real slot machine action than the TRS-90 version.

To summarize, if the program you want to convert is a number cruncher with a few graphics, the conversion should go smoothly and result in a TI BASIC program which runs with speed roughly comparable to its TRS-80 cousin. But if the program involves the heavy use of graphics, expect to rewrite it. And if the program is poorly documented to boot, keep a bottle of aspirin handy. Futhermore, because of the limitations of the TI BASIC IF-THEN-ELSE, and the lack of a PRINT AT command you can expect nearly every converted program to increase in length. On the plus side, however, the extended variable names available in TI BASIC make it possible to enhance the quality of the documentation and structure of the rewritten program.

One final note: TI's Extended BASIC Command Cartridge adds the PRINT AT and PRINT USING statements, has the capability of controlling up to 28 moving objects simultaneously, has improved IF-THEN-ELSE capability, and supports true subroutine definition (a significant aid in structuring programs). Although Extended BASIC probably won't alter the need for rewriting graphic programs, it should make the job a lot easier.

Language Conversion:

APPLESOFT to TI BASIC



The Apple II has also generated its fair share of applications and games programs—most of them taking advantage of the Apple's color graphics capability. In this regard the Apple is more like the TI-99/4A than the non-color TRS-80.

The APPLESOFT language card has about 29 non-graphic commands which are identical to TI BASIC. These commands, shown in Table 1 below, can be copied without much concern over compatibility.

ABS	DEF	GOTO	ON...GOSUB	SOP
ASC	DIM	INT	READ	STEP
ANT	END	LEN	REM	STOP
CHR$	EXP	LET	RETURN	STR$
COS	FOR...TO	LOG	SGN	TAN
DATA	GOSUB	ON...GOTO	SIN	

Table 1

In the remaining 26 or so commands, the differences range from very slight to major. Most importantly, the differences, though slight in format or content, can cause major problems in converting code. I'll go into each command, showing what to look for and how to resolve difficulties.

String Commands

APPLESOFT uses three different commands (LEFT$, MID$, and RIGHT$) in place of the TI's SEG$. The statement LEFT$(A$,N) references the first N characters of string A$. This directly translates into SEG$(A$,1,N). MID$(A$,M,N) is the same as SEG$(A$,M,N). Right$(A$,N) references the last N characters in string A$. The best way to duplicate this is to combine the LEN and SEG commands as follows: SEG$(A$,LEN(A$) – N + 1,N).

The VAL function acts the same way in both APPLESOFT and TI BASIC if the field being VALed is a valid numeric string. That is, both will return 45.2 as the value of "45.2". If the string does not contain valid numeric characters, however, the results are very different. TI BASIC will stop the program if the field contains non-numeric characters. APPLESOFT, however, will return with the numeric equivalent of the numbers found in the string before the first non-numeric character. For example: VAL ("123AB") will return with 123. If the first character of the string isn't numeric, APPLESOFT returns a 0.

This is important because it means that APPLESOFT does not have to edit a string prior to the VAL statement. A typical program will have code such as:

```
10 INPUT A$
20 X = VAL(A$)
30 IF X = 0 THEN 10
```

I've found that in most cases, I can ignore the whole issue by using TI's built-in numeric editor and coding INPUT X in place of statements 10 to 30 above. If you can't do this, use the following routine to replace the APPLESOFT VAL command:

```
10 FOR Y = 1 TO LEN (A$)
20 IF (ASC(SEG$(A$,Y,1))<48)
    + (ASC(SEG$(A$,Y,1))>57)THEN 40
30 NEXT Y
40 IF Y = 1 THEN 80
50 Y = Y – 1
60 Y = VAL(SEG$(A$,1,Y))
70 GOTO 90
80 Y = 0
90 END
```

Note: This is *not* a rigorous equivalent of APPLESOFT's VAL, but it is sufficient for whole numbers greater than – 1.

FOR-TO-STEP-NEXT

In the usual run of programs, the FOR-TO-STEP statement is identical in the two interpreters. There is, however, a very significant difference to look out for. The BASIC statement FOR Z = 5 TO 4 will execute once in APPLESOFT but will not execute at all in TI BASIC! This difference is important but can easily be spotted while transcribing a program. It isn't so obvious if the statement is FOR Z = A TO B where A and B are computed variables. The safest thing is to test for A greater than B. If it is, make B equal to A before entering the loop.

Both interpreters treat the STEP statement the same way and are very similar in the format of the NEXT statement—though in APPLESOFT, NEXT may be used by itself to end a single FOR loop. If the FOR loops are nested, however, APPLESOFT needs the control-variable name following NEXT, as does TI BASIC.

INPUT/OUTPUT (I/O)

Both machines use very similar INPUT and PRINT statements. They differ only in the use of print separators. Both use the comma as a tab command and the semicolon as a non-space separator. APPLESOFT reserves the colon

for a special use and doesn't treat it as a new line separator. When converting, always keep this in mind because it provides a powerful formating tool when converting PRINT statements. The TAB command is similar in both interpreters, but TI machine skips to a new line if a TAB value is less than the current column location. The APPLE will ignore the TAB statement in this case.

As part of the print function, APPLESOFT has a command of the format SPC(N), which is used to print N spaces. This must be replaced with a string of N spaces in the TI PRINT statement. APPLESOFT has to be very careful with spaces because it does not format a number with leading and trailing spaces the way TI BASIC does. This means that it is very rare to see something like PRINT J;K in APPLESOFT—a perfectly acceptable command in TI code since all numbers are printed with a trailing space.

The APPLE II screen starts off with the cursor at the top and works its way down to the bottom before scrolling begins. The APPLE uses HTAB and VTAB statements to shift the print position horizontally and vertically in order to print information at different locations on the screen. TI BASIC uses the colon, instead, to force line feeds. When converting, either change the print format to use line-feeds (colons), or use HCHAR to print at an equivalent location. Note: TI provides a full PRINT AT (using HCHAR) routine as part of its *Programming Aids I* package, but it is very slow. In many cases (where scrolling is acceptable), you are better off setting up a sequence of PRINT commands using the colon (PRINT : : : : : :). If you must use the HCHAR method of print out, here's a routine to print string A$ at row RO, column CO:

```
10 FOR X=1 TO LEN(A$)
20 CALL HCHAR
(RO,CO+X-1,ASC(SEG$(A$,X,1)))
30 NEXT X
```

This routine is much faster but requires you to remember to begin at column 3 (where TI BASIC begins its PRINT line) and not to allow A$ to extend past column 30 (where TI ends its PRINT line).

The prompt for APPLESOFT input is the same as for TI BASIC except that it uses a semicolon in place of the colon to separate the prompt from the input variable. For example:

```
10 "ENTER A NUMBER";Q
        VS
10 "ENTER A NUMBER":Q
```

The last I/O difference concerns getting a single character without using the INPUT statement: APPLE uses the GET statement, while TI uses the CALL KEY statement.

SCREEN COMMANDS

The APPLE has three modes of processing: Text mode and two different graphics modes. While in Text mode, the programmer has a number of commands which provide a wide range of control over the screen. The APPLE screen, in this mode, acts like the TI—except it starts at the top and works its way down to the bottom before scrolling. It also allows the programmer to set the width of the print screen ("text window") and the length (number of lines) of the text window, among other things. Some of the most commonly encountered commands are:

CALL -936	Clears the screen inside the test window
CALL -912	Scrolls the text window up 1 line
CALL -868	Clears the current line from the cursor to the right
HOME	Same as TI's CALL CLEAR
POKE 33,L	Sets left margin of window to L
POKE 33,W	Sets width of window
POKE 34,T	Sets top of window
POKE 35,B	Sets bottom of screen
FLASH	Starts 'flashing' output from white letters on black to black letters on white and back again
INVERSE	Reverses output to black letters on white
NORMAL	Resets FLASH and INVERSE
POS(N)	Gets current horizontal column of the cursor (i.e., N will have column number 0-39)

To simulate FLASH or INVERSE, use TI BASIC's CALL COLOR statement. For Example, CALL COLOR (3,16,2) gives white numbers from 0 to 7 on a black background. Changing this to CALL COLOR (3,2,16) will cause the inverse of it to appear (black numbers on a white background).

RANDOM NUMBERS

Because APPLESOFT has the ability to retain a random number for re-use, you cannot always convert the APPLE RND statement directly to TI. In APPLESOFT, if the statement is RND(0), APPLESOFT re-uses its last random number. If the statement is RND(N) where N is positive, it gives a new random number. If the statement is RND(N) where N is a negative number, N acts as a 'seed' number, and all other RND statements will follow a standard sequence. Note that the value N can be any positive number in order to give a new random number.

If you see a statement using RND(0), backtrack to the last statement with RND(N) and save that random number in place of RND(0). For example:

```
10 IF RND(2)<.5 THEN 500
:
60 IF RND(0)<.75 THEN 600
```

in APPLESOFT would convert in TI BASIC to:

```
10 Q=RND
15 IF Q<.5 THEN 500
:
60 IF Q<.75 THEN 600
```

MULTISTATEMENT LINES

A key point about APPLESOFT that I haven't yet mentioned is that it allows multiple statements on one program line. Each statement is separated by a colon. This allows code like:

```
10 X=X+Y:Y=Y+1:Z=Z+1
```

Translating multistatement lines can be a big problem because there may not be available line numbers to assign to the converted statement lines. For example:

```
400 A=A+1:FOR I=1 TO X:B=I*A:NEXT I
401 GOSUB 403
402 RETURN
403 REM
404 GOSUB 600
405 A=A+10
406 RETURN
```

The problem here is that there is no room to separate the multiple statements on line 400.

You can get around this by using a line number translation: Multiplying all line numbers by 10 allows you space to insert the extra line of code. The translated code is as follows:

```
4000 A = A + 1
4002 FOR I = 1 TO X
4004 B = I*A
4008 NEXT I
4010 GOSUB 4030
4020 RETURN
4030 REM
4040 GOSUB 6000
4050 A = A + 10
4060 RETURN
```

IF-THEN-ELSE

APPLESOFT does not require the ELSE feature of an IF statement because it allows other statements after the THEN part of the IF statement, as in the following:

```
10 IF A = X THEN X = X + 1:Y = Y + 1
20 A = X + Y
```

If X is equal to A, all statements following THEN are executed. If X isn't equal to A, the program simply advances to statement 20. The TI BASIC equivalent is:

```
10 IF X = A THEN 15 ELSE 20
15 X = X + 1
16 Y = Y + 1
20 A = X + Y
```

Because TI BASIC lacks multiple statements per line, it requires much more coding and a concurrent increase in memory needed for code. Keep this in mind if you are tempted to enter a program requiring 16K RAM in APPLESOFT; it probably won't fit in your TI machine. [Of course, if you have TI Extended BASIC, all this is moot, since this Command Cartridge allows multiple statement lines. See "HOW E-X-T-E-N-D-E-D IS EXTENDED BASIC?"—Ed.]

LOGICAL EXPRESSIONS

Both interpreters allow logical expressions to be used as if they were numeric values. APPLESOFT treats true expressions as if they are equal to 1, while false expressions are equal to 0. For TI BASIC true expressions are −1, false are 0. Whenever converting code from APPLESOFT, just insert a "−" in front of the logical expression:

```
10 X = (0$ = "A")*5
```

becomes

```
10 X = -(0$ = "A")*5
```

AND/OR

APPLESOFT allows multiple IF tests to be combined using the Boolean operators AND and OR. TI BASIC also allows this using the "*" and "+" arithmetic operators, respectively. For example:

```
10 IF (A = B) AND (C = D) THEN X = X + 1
```

is replaced with

```
10 IF (A = B)*(C = D) THEN 15 ELSE. . .
15 X = X + 1
```

In some cases, a straight conversion of the APPLESOFT IF-THEN will result in wasteful code. It is always a good idea to understand the purpose of the tests being made, and if possible, re-code them more efficiently. For example:

```
10 IF (A = B) AND (C = D) THEN X = X + 1
20 Y = Y + 1
```

would convert to:

```
10 IF (A = B)*(C = D) THEN 15 ELSE 20
15 X = X + 1
20 Y = Y + 1
```

but it would take less code (and therefore less core!) to invert the test:

```
10 IF (A <> B) + (C <> D) THEN 20
15 X = X + 1
20 Y = Y + 1
```

SPECIAL FUNCTIONS

Each interpreter has special functions oriented toward the manufacturer's hardware. Some of these are similar to other functions available in a different computer. I will list only the ones most commonly seen in APPLESOFT programs.

CLEAR	Initializes all variables. Automatically done by TI BASIC as part of RUN.
HIMEM LOMEM	Sets highest and lowest memory available to BASIC. No equivalent in TI BASIC.
FRE(0)	Gets amount of available memory left.
PDL(N)	GETS joystick input. In TI BASIC, use CALL JOYST instead. The PDL function returns with values from 0 to 255. If the value of N is 0 to 3, you are referencing the joysticks, but values from 4 to 255 can do weird things. Luckily, the APPLE joysticks don't seem to be used much. Also, the only way to test for the 'FIRE' buttons is to PEEK(−16287) through PEEK(−16284) for paddles 0 thru 3.
POP	Cancels the last GOSUB. This is mostly used in edit subroutines where an error causes the progam to go to an error routine instead of RETURNing. The only way to code an equivalent in TI BASIC is to have the edit routine coded in an error switch which is interrogated as soon as the subroutine RETURNs.
ON ERR RESUME	This tells APPLESOFT to GOTO a part of the program if it encounters certain errors while processing. In TI BASIC, any errors are either handled by the BASIC interpreter (e.g., dividing by zero), or cause the program to end (e.g., reading past the last DATA statement). The ON ERR is most often used to trap an error expected by, or consciously caused by the programmer.
USR(X)	Jump to a machine language subroutine.

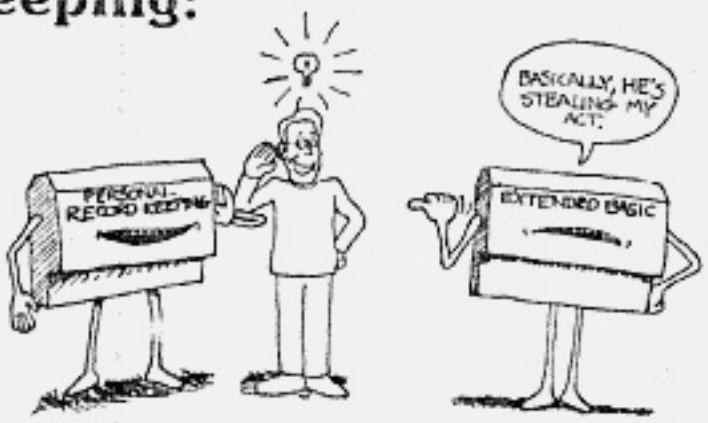
As you can see from the foregoing, converting most code from APPLESOFT to TI BASIC is straightforward, with most of the effort devoted to converting PRINT statements. Most importantly, don't get frustrated if your first attempts don't succeed the way you intended. After a while, it will all become second nature.

The Secret of Personal Record Keeping:

Implementing

DISPLAY AT and ACCEPT AT

Without Extended BASIC



Some of you may have accidentally stumbled upon features of the TI-99/4 that are not described anywhere but which are nonetheless quite helpful. I did. . .and what happily resulted was a way to quickly print text to and accept it from anywhere on the screen without having to pass through loops or causing the screen to scroll.

Those of you with Extended BASIC already have this capability with the DISPLAY AT and ACCEPT AT statements. Now you can have these powerful features in TI BASIC (the language built into the TI-99/4 and 99/4A computers), provided the *Personal Record Keeping* Command Cartridge is inserted. This cartridge, which is quite powerful and versatile in itself, will interface with the console's BASIC routines and allow you to use two new statements: CALL D and CALL A. [See "Personal Record Keeping: Managing a Mobile Home Park" for more information on the *PRK* cartridge. Those of you without the *PRK* cartridge but who happen to have the *Statistics* cartridge should be able to use that instead.—Ed.]

Before getting into the documentation, I should, of course, mention that you can also print anywhere on the screen without CALL D by handling the printing character by character using the subroutine given in the examples in your manuals, i.e., "Character Definition." The drawbacks of that method include lack of speed (the letters appear one by one), more cumbersome programming and more memory space taken up.

1. DISPLAY AT – numerical data

CALL D (R, C, L, V)

R = row number of first character of print line
C = column number of first character of print line
L = maximum length of print line; must be >=1
V = variable for the value that is to be printed

R/C— The R(ow) and C(olumn) variables are meaningful with values between 1 and 24, and 1 and 28, respectively (the print field 24 x 28 is used). Values below the minimum of 1 (0 and negative numbers) are treated as the value 1. Values above the maximum (24 or 28) are automatically subtracted as many times as is required to bring the result between 1 and 24 or 28; this result is then used as the R and C value. This is a nice feature that eliminates many program halts of "BAD VALUE" that often result from careless programming. Data at the end of the screen line is not printed at the beginning of the next screen row as is the case with the CALL HCHAR statement.

L— The L position can be used with a fixed number (the maximum meaningful number is 28) or as a variable to which the function can be assigned in numerical form, like SEG$ in strings.

V— Instead of a numerical variable, you can also put a number in this position; it will then be printed on the screen in a position according to the rules above.

Example 1

```
100 CALL CLEAR
110 V=326525
120 CALL D(12, 10, 5, V)
130 GOTO 130
```

Of course you can explain why this program displays only 3265 in the middle of the screen. (Remember that a sign—equivalent to a digit—precedes each number, and that plus signs are suppressed on printing.) How would you have to change line 120 to give the full 326525?

2. DISPLAY AT - string data

Version 1: CALL D(R, C, L, S$)
Version 2: CALL D(R, C, L, ("PAUL W. KARIS")
Version 3: CALL D(R, C, L, CHR$(N))

The variables R, C, and L work as described previously under section 1, above.
Here expecially, L can be put to good use as a built-in SEG$.

ersion 1: the string variables S$ is printed
ersion 2: the string between quotes is printed
ersion 3: a complicated way of saying CALL HCHAR(R,
, N) that is merely mentioned here as illustration of the
ssiblities

xample 2
100 CALL CLEAR
110 A$="THIS IS MID-SCREEN"
120 CALL D(12, 4, 19, A$)
130 GOTO 130

ACCEPT AT – numerical data

The ACCEPT AT statement works like INPUT but can
: formated anywhere on the screen. The input prompt can
: printed in the appropriate place with the technique of
ction 2, above. The built-in value checks are an additional
ature.

ALL A(R, C, L, F, A, MN, MX)

, C, and L have been explained in section 1.

= function variable
= accept variable
IN = minimum value
IX = maximum value

F— The numerical variable in this position assumes a value 1-7 depending on certain function keys being depressed. The values connected to these functions in this way should not be confused with the ASCII values of these functions that can be useful in CALL KEY statements. For completeness, I'll also tabulate the ASCII values here.

Function Key		CALL A value (F position)	ASCII value
TI-99/4A	TI-99/4		
FCTN 5	SHIFT W – BEGIN	6	14
FCTN 8	SHIFT R – REDO	4	6
FCTN 7	SHIFT A – AID	3	1
FCTN 9	SHIFT Z – BACK	7	15
FCTN 4	SHIFT C – CLEAR	2	2
FCTN 6	SHIFT V – PROC'D	5	12
	ENTER	1	13

CLEAR will not only give F a value of 2, but it also clears the input printing field on the screen and is to be used when typed input is not yet entered and should be changed. Warning: This means that if you write a program that continually loops to a CALL A statement, CLEAR cannot be used to break the program. Only QUIT or cutting the power will work then, but it will also erase your program in the process! The solution to this problem is to program your escape routine, e.g., IF F=3 THEN 10000 enabling you to use AID to bring the program to line 10000 which reads: 10000 END.

A— The variable in the position of A assumes (accepts) the value you typed in much in the same way as the input variable does after you depress ENTER. The F variable, of course, then gets the value 1 since you have used the function key ENTER. If you press ENTER when the print/input field contains no information (only "space"), F will take on the value in the above table if one of the function keys has previously been pushed.

MN— The numbers or the values of the numerical
MX— variables in the positions MN and MX respectively determine the minimum and maximum values that A will accept. A gentle beep when you press the ENTER warns you if you try to step beyond these imposed limits. The screen, of course, will accept any numerical data, provided that the length does not exceed L(e.g., if L=2 and MX=10000 you still cannot get A to become more than 99 since the screen will not accept more than 2 digits). Since the plus and minus signs (+ and –) as well as the letter E (scientific notation) are all considered to be numerical input, they will also be accepted. String data, however, are not accepted by the screen at all when you use CALL A in this way.

If MN=MX, A will accept only the MN and the MX value. If MN>MX, A shouldn't accept any value at all, but illogically, it does accept the MN value.

Example 3

```
100 CALL CLEAR
110 CALL D(3, 3, 28, "ENTER 1, 2, OR 3")
120 CALL A(10, 25, 1, F, B, 2, 3,)
130 CALL CLEAR
140 FOR T=1 TO 500
150 NEXT T
160 CALL D(15, 3, 28, "YOUR CHOICE WAS")
170 CALL D(15, 20, 2, B)
180 FOR T=1 TO 500
190 NEXT T
200 GOTO 100
```

4. ACCEPT AT – string data

CALL A(R, C, L, F, A$)

R, C, and L are explained in section 1.
F is explained in section 3.
A$ = accept string variable.

A$ The variable in the A$ position is filled with the typed string information when you press ENTER.

Example 4

```
100 CALL CLEAR
110 M$ ="PLEASE ENTER YOUR NAME"
120 CALL D(5, 3, 26, M$)
130 CALL A(10, 3, 20, F, N$)
140 CALL CLEAR
150 FOR T=1 TO 500
160 NEXT T
170 CALL D(5, 2, 28, "THANKS " & N$)
180 FOR T=1 TO 500
190 NEXT T
200 GOTO 100
```

Now you're on your own: It's your turn to apply these two new commands and, perhaps, discover some additional ones.

[Note: In the event that Texas Instruments gets away from producing "hybrid" Command Cartridges (containing both BASIC and GPL coding), future releases of *Personal Record Keeping* will not offer the capabilities described in this article.—Ed.]

Dynamic Screen Graphics

MANIPULATION OF CHARACTER

Would you appreciate being able to write shorter programs that effectively do the same thing as longer ones? Or, would you enjoy watching the computer do a large amount of the tedious and boring designing, defining and selecting of dozens of graphics characters—work that you would otherwise have to do yourself? If your answer to both of these questions is YES, read on, fellow 99'er.

The scheme used in the TI-99/4A to represent screen character patterns with hexadecimal numbers is compact and convenient—ingenious really. It's compact because only 16 digits uniquely specify the on-off states of the 64 pixels in each 8 × 8 pixel character block. Such a system is certainly more satisfactory than display systems that provide only a small selection of predefined characters. It's convenient because the programming requires only simple statements of the form:

```
CALL CHAR(IJK,"0123456789ABCDEF")
```

to define any 8 × 8 character imaginable. Likewise the statement:

```
CALL HCHAR(ROW,COLUMN,IJK,REPEAT)
```

will put character IJK anywhere on the screen. After a brief period, one is able to work intuitively, giving little conscious thought to the format.

Yet even with this system, there remains a considerable amount of tedious work to be done because every character we want on the screen (beyond the resident alphabet, etc.) must be defined and must be located. Doing this for many characters can mean lots of work, as in Figure 1, where a graphic occupying less than half the screen contains 33 different characters. All 64 user-definable characters would use up 64 lines of code just to define; if resident characters were redefined, we could end up having in memory a hundred or so program lines devoted to this one purpose.

In addition, there is the wear and tear on the programmer. He gets his ears burned if he leaves out one of those quote marks. Additional possibilities for errors include leaving out a comma or parenthesis or, worse, having a pattern identifier string with more or less than 16 numbers, or inadvertently typing in a nonhexadecimal symbol. Just type in four or five dozen CALL CHAR(IJK,"0123456789 ABCDEF") statements and you will surely develop an acute case of boredom. Such static definition—with a program line for every new character and the resulting long list of CALL CHAR statements—is a lot of trouble and a source of errors.

It is also unnecessary. A little experimenting will show that we can define screen characters with data statements and a loop. Only a single CALL CHAR statement need be

yped in and carried in memory. Such a method was used
the program which draws Figure 1. The program is given
Listing 1, *Xmas-Tree*. The hexadecimal strings which
efine the screen characters to be used are in data statements
arting at line 270. The loop starting at line 440 reads a
ata statement and puts the hexadecimal string it has pick-
d up into a CALL CHAR statement. Thus the definition
sent off to graphic memory where it can be used later
the program as many times as needed. In this program,
ach data entry contains a comment to help one figure out
hat is happening on the screen, and each data entry con-
ins three items: identification string, character number,
nd pattern-identifier string. On the next pass through the
oop, another hexadecimal string is picked up and put in
he CALL CHAR statement. Thus another defined screen
haracter is sent off to memory.

After the program has cycled the last time through the
oop, all the screen characters described in the data
tatements are in memory. They are now available using
CALL HCHAR or CALL VCHAR statements just as if
he program had run through dozens of CALL CHAR lines.
ewer program lines have been used, the possibility of er-
ors reduced, and life has been made much easier for the
rogrammer.

In a similar manner, characters are located on the screen
eginning at line 740. For this application the data entries
ave the form: identification string, row number, column
umber, character number. The identification string serves
nly as documentation. The loop at line 940 puts this in-
ormation in a CALL HCHAR statement which then sends
off to the video display processor. All characters will now
ppear on the screen at their assigned locations. Of course,
he information we have in data statements could also be
tored on a floppy disk.

Dynamically defining characters and putting them on the
creen with data statements and loops (1) saves program lines
nd effort, (2) reduces errors, and (3) can make a program
asier to follow if documentation is added. Although for
his program no special attempt has been made to reduce
he memory required, the information in data statments
ould be packed tighter by omitting identification. Also, we
ould incorporate the number of repetitions in the data
tatements.

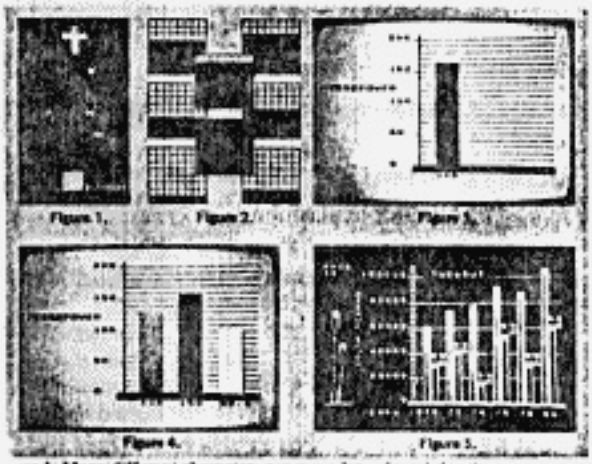


Figure 1.

Figure 2.

Figure 4.

Figure 5.

ure 1. Many different characters can mean lots of work for the programmer.
re 2. Screen characters used for one-pixel resolution in bar height.
3. Bar graph with one-pixel resolution.
4. Three variables plotted with one-pixel resolution.
5. An example of 99/4 graphics.

Another opportunity for making character definition and placement a part of program dynamics occurs in plotting bar graphs. Bar graphs are a frequent application for computer graphics, and they look terrific on the color monitor.

On the TI-99/4A it is easy to plot a bar (Y characters high) by just using CALL VCHAR(ROW,COLUMN,IJK,Y). But the resolution will be very poor because we can adjust the bar height in increments of only one full character, which is about 3/8 of an inch on the 13-inch monitor. Ideally we'd have a continuously adjustable bar height, but this infinite resolution cannot be realized with raster-scan systems. We can, however, get resolution equal to the pixel height. Toward this end we will define eight screen characters as shown in Figure 2. The first character has the bottom row of pixels turned on, the next one has the bottom two rows turned on, etc. The eighth character has all pixels turned on.

These characters are then used as bar tops. Stick the right one on top of your bar graph and you have resolution of one pixel (which is 1/8 of a character)—quite satisfactory with existing CRT's. On the 13-inch monitor this height increment is about 3/64 of an inch.

The program in Listing 2, *Bar-Topper*, which uses this method, plots the bar graph in Figure 3. The characters available for use as bar tops are defined beginning at line 360. Scale of 1 character = 10 units is applied to the value entered at the keyboard starting at line 700. The integral value of Y is found and the remainder used to select the bar top character needed. The actual selection is done by the ON GOTO statement at line 780.

This program does work, but represents a brute force approach. If there is only one bar on the graph, then only one character will be used at the bar top. Yet eight bar-top characters have been defined and are sitting in memory. To take an extreme case, suppose we have four variables to be represented by four bars of different colors. Here, 32 characters must be defined and available for use as bar tops, yet only four bar-top characters will actually be used. Besides taking up memory, we have used half of the user-defined characters. This approach is wasteful. Why define characters that sit in memory but are never used?

Let's try a better idea by devising a program that defines bar-top characters after reading the data. Then it can define only characters that are needed. In other words, the data determine what bar-top characters are defined. To do this, we will have in the program a master string containing fourteen zeros and sixteen F's. Segments exactly sixteen spaces long can be taken from this master string with a SEG$ statement. Next, the segment can be used as the pattern-identifier string and put in a CALL CHAR statement to define a bar top. Where will these 16-space segments start? Well, the data can cause a character with the first row of pixels turned on to be defined, or a character with the second row turned on, etc.

A possible coding to do this might be as follows:

```
110 MASTER$ = "00000000000000FFFFFFFFFFFFFFFF"
115 REMAINDER = BARHEIGHT - INT(BARHEIGHT)
120 TOPPATTERN = INT(REMAINDER*8 + .5) + 1
130 STARTPOSITION = 2*TOPPATTERN - 1
140 TOPPATTERN$ = SEG$(MASTER$,STARTPOSITION,16)
150 CALL CHAR(97,TOPPATTERN$)
160 CALL HCHAR(21 - Y,16,97,3)
```

Here the 21 in 21 – Y allows the bar to be up to 20 rows high.

Suppose, for example, that data calls for a bar top with the bottom two rows turned on. Then TOPPATTERN will

be 2. Then STARTPOSITION = 3. Then the pattern-identifier string created in line 140 will be

TOPPATTERN$ = "0000000000000FFFF"

(as you can see, if you will take the trouble to count this off, starting at the third position in the master string). The resulting screen character that is defined in line 150 will be one with the bottom two rows of pixels turned on. As the program runs, we want each datum to determine where the 16-space segment will begin. Thus we have used the remainder to calculate STARTPOSITION. By notching back and forth with STARTPOSITION, the routine will define any character needed to top off a bar.

With this particular routine there will be a little problem associated with rounding up to the next higher grid line on the next higher row. For instance, if the scale used is 1 character = 10 units, we would want 99.9 to appear on the graph as 100. Another problem (I didn't say this was too simple) involves the character to be used for the body of the bar. This character must have all pixels turned on, but the routine above will not create such a character for all values of the data set.

Auto-Top, a program in which these problems are solved, is given in Listing 3. A routine similar to the one above starts on line 750. Character 96, which is used for the body of the bar, is defined earlier in the program. Note that this master string contains 18 F's. (If you try this program, you had better count them carefully.) TOPPATTERN = 9 will pick up the extra F's at the 17th and 18th positions.

The problem of rounding up to the next higher grid line (so 99.9 will show up as 100 as in the earlier example) is taken care of in lines 820 and 830 where a one-row-on character is defined and put on the very top of the bar if, and only if, TOPPATTERN = 9.

A graph with only one bar is not very useful. We can generate additional bars with a loop. The routine in Listing 4, *Three-Bars*, plots three bars of different colors. See line 680. (My 13-inch monitor displays a lot of spillover with most colors—especially with red. There is less spillover with light or medium green or blue, and with white and yellow.) As the loops runs, it will shift to succeeding color sets with the expression 89 + BAR*8 as can be deduced by considering the statement

CALL CHAR(89 + BAR*8,TOPPATTERN$).

When BAR = 1, this statement defines character 97; when BAR = 2, character 105; and when BAR = 3, character 113. The first character is in color set 9, the second in color set 10, and the third in color set 11, allowing for three bars of different colors.

The position of the bars is shifted by the expression. 11 + 5 = 16 is the position of the left edge of the first bar, and the left edges of all bars are 5 columns apart. These bars are three columns wide. Figure 4 shows this graph as photographed on the 13-inch monitor.

This program and the earlier ones here might be a little longer than if they were written in the standard way. However, they will not get much longer if the graphics are made more elaborate. For example, the bar graph program does not get much longer if more bars are added.

The bar graph in Figure 5 was made using these techniques. I present it here just to show off the kind of goodlooking graphics that can be made with the TI-99/4A and TI BASIC. This program—with its outlining and the fact that it reads and writes data for eight variables from files and calculates items such as percentages—is more involved than the listing given here.

This brings up a new problem that has been created: In many of my programs I run out of characters. I did not notice this limitation when I was typing in so many CALL CHAR, CALL HCHAR, and CALL VCHAR statements. Actually when you think about it, there are not very many characters available. If you start at the left of the screen and put a different character in each space, you will run out of characters in the fifth line if you include punctuation, number, the alphabet, and the eight user-definable sets.

In other words, it takes only about 17% of the screen to display all available characters. Mathematically, we are not about to run out of characters since there are 256 different ways to put together just one row of a character. And the number of characters that can be on the screen in this graphic mode is 24 rows of 32 columns = 768 spaces.

Since my interest is primarily in graphics, available user-definable characters are more important to me than memory. Memory problems can often be avoided. To put a unique character on every space on the screen would require 48 character sets—several times more than any home computer presently has. I do not know if this is unreasonable. Two years ago the idea of a 48K memory sounded unreasonable. Perhaps some computer architect will devise a method of going to a higher resolution with nested character sets. [For a discussion of the high-resolution bit-mapped graphics supported by the TI-99/4A, see "3 – D Animation with the TMS9918A Video Chip."—Ed.]

Finally, note that for some applications it can be useful to define random graphics characters. This process, however, really eats up character sets. In Listing 5, *Twinkle*, random characters are defined that also have a certain amount of shape. Line 240 of this code generates random numbers from 1 to 16, and lines 480 to 620 convert them to hexadecimal notation 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. These numbers are assembled into a 16-space string. This hexadecimal string then goes into a CALL CHAR statement to define a random graphic character.

Shape is forced on the character in lines 280 to 470 by rejecting certain numbers generated by the random number generator. In this particular application, the edges of the characters are "rounded off" so they will not appear square.

I use such random-patterned screen characters to soften up the edges of my "block graphics" designs. ("Blockhead graphics?") Another application is to create dramatic effects as is done in *Twinkle* given in Listing 5.

I also use random characters to induce variations on things that, as in nature, change with time-shadows or explosions, for instance. Some video games could undoubtedly profit from this technique. I get a little tired of aliens that always blow up the same way. Hmm—come to think of it, there is that video game with the pigeon in it. . . .

Listing 1

```
100 REM  *-*-*-*-*-*-*-*-*-*-*-*-*
110 REM  *     XMAS-TREE     *
120 REM  *-*-*-*-*-*-*-*-*-*-*-*-*
130
140
150 REM  ABOUT 7568 BYTES
160 REM    PRESS ANY KEY TO END SCREEN
    DISPLAY.
170 CALL SCREEN(2)
180 CALL COLOR(9,7,1)
190 CALL COLOR(10,14,1)
200 CALL COLOR(11,14,4)
210 CALL COLOR(12,12,4)
220 CALL COLOR(13,2,4)
230 CALL COLOR(14,7,4)
240 CALL COLOR(15,15,1)
250 CALL COLOR(16,5,16)
260 REM  PATTERN-IDENTIFIERS
270 REM  FORMAT: IDENTIFICATION, CHARA
    CTER NUMBER, HEXADECIMAL STRING...
280 REM  EXAMPLE: TREE TRUNK,142,4C53
    8C3C659487A, TREE BODY,143,000000
    000000000...
290 DATA LEFTSIDE OUT,96,0107070F0F0F1
    F7F,RIGHTSIDE OUT,97,80E0E0F9F0F0F
    8FE,BORDERTOP,98,7F7F3F070701010C0
300 DATA TREE BOTTOM,99,FFF8F8F0F0E0C0
    80,8,100,FF3F3F0F0707030100
310 DATA BORDER TOP,102,FFFFFFFFFF0000
    00,BORDER BOTTOM,105,000000FFFFFFFF
    FFF
320 DATA PLUM,104,10FEFEFEFE7C3810,BEL
    L,107,1010383838C7C7CFE
330 DATA PLUM,112,10FEFEFEFE7C3810,DIA
    MOND,113,1010387CFE7C3810,BELL,115
    ,1010383838C7C7CFE
340 DATA PLUM,120,10FEFEFEFE7C3810,DIA
    ,121,1010387CFE7C3810,BELL,123,101
    038383C7C7CFEFE
350 DATA PLUM,128,10FEFEFEFE7C3810,DIA
    MOND,129,1010383C7CFE7C3810,BELL,131
    ,1010383838C7C7CFE
360 DATA LEFT INSIDE,136,FEF8F0F0E0E0C
    000,RIGHT INSIDE,137,7F1F0F0F070703
    00,CT,138,FFE7E7C3810101
370 DATA BOTTOM IN,150,0107070F0F0F3FF
    F,8,140,80C0E0E0F0FCFF
380 DATA TRUNK,142,4C53C659487A,TR
    EE BODY,143,000000000000000,POT L
    ,144,3F3F3F3F3F3F3F3F
390 DATA POT LEFT BOTTOM,146,3F3F3F3F3
    F3F0F0F,POT R,147,FCFCFCFCFCFCFCFC
400 DATA POT R B,149,FCFCFCFCFCFCFCF0,
    P B,150,FFFFFFFFFFFFFFFF
410 DATA TOP,152,C3C33C18183CC3C3,STAR
     RADIAL,155,FFFFFF0000FFFFFF,STAR
    RADIAL,154,E7E7E7E7E7E7E7E7
420 REM  DEFINE-LOOP
430 RESTORE 290
440 FOR CODE=96 TO 154
450 READ IDENTIFICATIONS,CHARACTERNUMB
    ER,HEX$
460 IF CHARACTERNUMBER>CODE THEN 480
470 GOTO 490
480 CODE=CHARACTERNUMBER
490 CALL CHAR(CODE,HEX$)
500 NEXT CODE
510 REM  START SCREEN DISPLAY
520 CALL CLEAR
530 REM  -----TREE BODY-----
540 CALL HCHAR(24,1,145,32)
550 CALL HCHAR(19,6,143)
560 CALL HCHAR(18,2,143,11)
570 CALL HCHAR(17,3,143,9)
580 CALL HCHAR(16,3,143,8)
590 CALL HCHAR(15,4,143,7)
600 CALL HCHAR(14,4,143,7)
610 CALL HCHAR(13,4,143,6)
620 CALL HCHAR(12,5,143,5)
630 CALL HCHAR(11,6,143,4)
640 CALL HCHAR(10,6,143,3)
650 CALL HCHAR(9,6,143,3)
660 CALL HCHAR(8,7,143)
670 REM  -----TREE TRUNK-----
680 CALL VCHAR(20,7,142,2)
690 REM  -----PLANT POT-----
700 CALL VCHAR(22,6,144,2)
710 CALL VCHAR(22,7,159,3)
720 CALL VCHAR(22,8,147,2)
730 REM  SCREEN LOCATION DATA
740 REM  FORMAT:IDENTIFICATIONS,ROW,CO
    LUMN,CHARACTERNUMBER...
750 DATA POT BASE LEFT SIDE,24,6,146,P
    OT BASE RIGHT SIDE,24,8,149
760 REM  -----FOLLAGE-----
770 DATA LO,18,1,96,LI,18,2,136,LO,17,
    1,96,LI,17,2,136,LO,16,1,96,LI,16,
    2,136,LO,15,2,96,LI,15,3,136
780 DATA LO,14,2,96,LI,14,3,136,LO,13,
    2,96,LI,13,3,136,LO,12,3,96,LI,12,
    4,136,LO,11,4,96,LI,11,5,136
790 DATA LO,10,4,96,LI,10,5,136,LO,9,4
    ,96,LI,9,5,136
800 DATA LO,5,6,96,CT,5,7,138,LO,6,6,9
    6,LI,6,7,136,LO,7,5,96,LI,7,6,136,
    LO,8,5,96,LI,8,6,136
810 DATA RO,5,8,97,RO,6,8,97,RI,6,7,13
    7,RO,7,8,97,RI,7,7,137,RO,8,9,97,R
    I,8,8,137,RO,9,10,97
820 DATA RI,9,9,137,RO,10,10,97,RI,10,
    9,137,RO,11,11,97,RI,11,10,137,RO,
    12,11,97,RI,12,10,137
830 DATA RO,13,11,97,RI,13,10,137,RO,1
    4,12,97,RI,14,11,137,RO,15,12,97,R
    I,15,11,137,LO,16,12,97
840 DATA LI,16,11,137,LO,17,13,97,LI,1
    7,12,137,LO,18,13,97,B,18,12,137,B
    ,19,12,99,B1,19,11,139
850 DATA B1,19,10,140,LO,19,9,100
860 DATA BOTTOM,20,5,96,B,20,6,99,B,19
    ,7,139,B,19,8,99,B,19,4,100,B,19,5
    ,140,B,19,2,96,B,19,3,99
870 DATA B IN,18,7,159,B OUT,18,8,136
880 REM  -----CROSS-----
890 DATA TOP,2,7,152,L RADIAL,2,6,155,
    R RADIAL,2,8,155,T RADIAL,1,7,154,
    B RADIAL,3,7,154,B RAD,4,7,154
900 REM  -----ORNAMENT-----
910 DATA OUTSIDE BELL,7,9,104,PLUM,20,
    2,107,PLUM,14,4,112,DIAMOND,13,9,1
    13,BELL,16,10,115
920 DATA PLUM,13,4,112,PLUM,12,9,128,D
    IAMOND,14,4,129,BELL,17,3,131
930 REM  SCREEN LOCATION LOOP
940 HOWMANY=86
950 RESTORE 750
960 FOR CHARACTER=1 TO HOWMANY
970 READ IDENTIFICATIONS,ROW,COLUMN,CH
    ARACTERNUMBER
980 CALL HCHAR(ROW,COLUMN,CHARACTERNUM
    BER)
990 NEXT CHARACTER
1000 CALL KEY(0,K,S)
1010 IF S=0 THEN 1000
1020 END
```

Listing 2

```
100 REM  *-*-*-*-*-*-*-*-*-*-*-*-*
110 REM  *  BAR-TOPPER  *
120 REM  *-*-*-*-*-*-*-*-*-*-*-*-*
130 REM
140 REM
150 REM
160 REM  PRESS ANY KEY TO STOP DISPLAY
    .
170 VERTICALMAX=200
```

```
100 SCALE=VERTICALMAX/20
110 CALL CLEAR
120 LABELS="ENTER HORSEPOWER"
130 ROW=12
140 COLUMN=15
150 GOSUB 1000
160 LABELS="0 TO 200"
170 ROW=13
180 COLUMN=19
190 GOSUB 1000
200 INPUT "":HORSEPOWER
210 CALL SCREEN(8)
220 CALL COLOR(9,13,8)
230 CALL COLOR(10,2,5)
240 REM DEFINE CHARACTERS
250 REM FORMAT:IDENTIFICATIONS,CHARA
    CTERNUMBER,HEXADECIMALS...
260 REM ---GRID---
270 DATA GRID LINE,91,00000000000000FF
    ,VERTICAL AXIS,92,0101010101010101
    ,TIC MARK,93,01010101010101FF
280 REM --DEFINE BAR TOPS--
290 DATA BOTTOM ROW OF PIXELSON,96,000
    00000000000FF,SECOND ROW ON,97,000
    0000000000FFFF,THIRD ROW ON
300 DATA 98,000000000FFFFFF,FOURTH,99
    ,0000000FFFFFFFF,FIFTH,100,000000
    FFFFFFFFFF,SIXTH,101,0000FFFFFFFF
    FFFF
310 DATA SEVENTH,102,00FFFFFFFFFFFFFF,
    EIGHTH,103,FFFFFFFFFFFFFFFF
320 REM ---BASELINE---
330 DATA BASE,104,FF0000FF000000FF
340 REM DEFINE LOOP
350 RESTORE 270
360 FOR CODE=91 TO 104
370 READ IDENTIFICATIONS,CHARACTERNUMB
    ER,HEXS
380 IF CHARACTERNUMBER>CODE THEN 480
390 GOTO 490
400 CODE=CHARACTERNUMBER
410 CALL CHAR(CODE,HEXS)
420 NEXT CODE
430 REM START SCREEN DISPLAY
440 REM ---GRAPH GRID---
450 CALL HCHAR(22,13,104,18)
460 FOR ROW=21 TO 1 STEP -1
470 CALL HCHAR(ROW,14,91,17)
480 NEXT ROW
490 LABELS="HORSEPOWER"
500 ROW=9
510 COLUMN=1
520 GOSUB 1000
530 CALL VCHAR(1,13,92,21)
540 FOR ROW=21 TO 1 STEP -5
550 ROWNUMBER=200-(10*(ROW-1))
560 LABELS=STR$(ROWNUMBER)
570 COLUMN=10
580 GOSUB 1000
590 CALL HCHAR(ROW,13,93)
600 NEXT ROW
610 REM CALCULATE BAR HEIGHT
620 BARHEIGHT=HORSEPOWER/SCALE
630 Y=INT(BARHEIGHT)
640 REMAINDER=BARHEIGHT-INT(BARHEIGHT)
650 CALL VCHAR(22-Y,16,103,Y)
660 CALL VCHAR(22-Y,17,103,Y)
670 CALL VCHAR(22-Y,18,103,Y)
680 REM SELECT BAR TOP
690 TOPPATTERN=INT((8*REMAINDER+8)+.5)
700 ON TOPPATTERN+1 GOTO 790,810,830,8
    50,870,890,910,930,950
710 CALL HCHAR(21-Y,16,96,3)
720 GOTO 970
730 CALL HCHAR(21-Y,16,97,3)
740 GOTO 970
750 CALL HCHAR(21-Y,16,98,3)
760 GOTO 970
770 CALL HCHAR(21-Y,16,99,3)
780 GOTO 970
790 CALL HCHAR(21-Y,16,100,3)
800 GOTO 970
810 CALL HCHAR(21-Y,16,101,3)
820 GOTO 970
830 CALL HCHAR(21-Y,16,102,3)
840 GOTO 970
850 CALL HCHAR(21-Y,16,103,3)
860 GOTO 970
870 CALL HCHAR(21-Y,16,100,3)
880 GOTO 970
890 CALL HCHAR(21-Y,16,101,3)
900 GOTO 970
910 CALL HCHAR(21-Y,16,102,3)
920 GOTO 970
930 CALL HCHAR(21-Y,16,103,3)
940 GOTO 970
950 CALL HCHAR(21-Y,16,103,3)
960 CALL HCHAR(20-Y,16,96,3)
970 CALL KEY(0,K,S)
980 IF S=0 THEN 970
990 END
1000 FOR POSITION=1 TO LEN(LABELS)
1010 LETTERS=SEGS(LABELS,POSITION,1)
1020 CODE=ASC(LETTERS)
1030 CALL HCHAR(ROW,COLUMN-1+POSITION,C
     ODE)
1040 NEXT POSITION
1050 RETURN
```

Listing 3

```
100 REM  *+*+*+*+*+*+*+*
110 REM  *  AUTO-TOT  *
120 REM  *+*+*+*+*+*+*+*
130 REM
140 REM
150 REM  ABOUT 5288 BYTES
160 REM  PRESS ANY KEY TO STOP DISPLAY
170 VERTICALMAX=200
180 SCALE=VERTICALMAX/20
190 CALL CLEAR
200 LABELS="ENTER HORSEPOWER"
210 ROW=12
220 COLUMN=15
230 GOSUB 870
240 LABELS="0 TO 209"
250 ROW=13
260 COLUMN=19
270 GOSUB 870
280 INPUT "":HORSEPOWER
290 CALL SCREEN(8)
300 CALL COLOR(9,13,8)
310 CALL COLOR(10,2,5)
320 REM DEFINE CHARACTERS
330 REM FORMAT:IDENTIFICATIONS,CHARAC
    TERNUMBER,PATTERNS...
340 DATA GRID LINE,91,00000000000000FF
    ,VERTICAL AXIS,92,0101010101010101
    ,TIC MARK,93,01010101010101FF
350 DATA BAR,96,FFFFFFFFFFFFFFFF,BASEL
    INE,104,FF0000FF000000FF
360 DATA RESERVED FOR TITLE BOX
370 DATA RESERVED FOR LABELS
380 DATA RESERVED FOR LEGEND
390 DATA RESERVED FOR ADDITIONAL CHARA
    CTERS
400 REM DEFINE-LOOP
410 RESTORE 340
420 FOR CODE=91 TO 104
430 READ IDENTIFICATIONS,CHARACTERNUMB
    ER,PATTERNS
440 IF CHARACTERNUMBER>CODE THEN 460
450 GOTO 470
460 CODE=CHARACTERNUMBER
470 CALL CHAR(CODE,PATTERNS)
480 NEXT CODE
490 REM START SCREEN DISPLAY
500 REM ---GRAPH GRID---
510 CALL HCHAR(22,13,104,18)
520 FOR ROW=21 TO 1 STEP -1
530 CALL HCHAR(ROW,14,91,17)
540 NEXT ROW
550 LABELS="HORSEPOWER"
560 ROW=9
570 COLUMN=1
580 GOSUB 870
```

```
590 CALL VCHAR(1,13,92,21)
600 FOR ROW=21 TO 1 STEP -5
610 ROWNUMBER=200-(10*(ROW-1))
620 LABEL$=STR$(ROWNUMBER)
630 COLUMN=10
640 GOSUB 870
650 CALL HCHAR(ROW,13,93)
660 NEXT ROW
670 REM  CALCULATE BAR HEIGHT
680 BARHEIGHT=HORSEPOWER/SCALE
690 Y=INT(BARHEIGHT)
700 REMAINDER=BARHEIGHT-INT(BARHEIGHT)
710 CALL VCHAR(22-Y,16,96,Y)
720 CALL VCHAR(22-Y,17,96,Y)
730 CALL VCHAR(22-Y,18,96,Y)
740 REM  SELECT BAR TOP
750 TOPPATTERN=1+INT((REMAINDER*8)+.5)
760 MASTER$="0000000000000000FFFFFFFFFFFF
    FFFFFF"
770 STARTPOSITION=2*TOPPATTERN-1
780 TOPPATTERN$=SEG$(MASTER$,STARTPOSI
    TION,16)
790 CALL CHAR(97,TOPPATTERN$)
800 CALL HCHAR(21-Y,16,97,3)
810 IF TOPPATTERN<>9 THEN 840
820 CALL CHAR(98,"0000000000000000FF")
830 CALL HCHAR(20-Y,16,98,3)
840 CALL KEY(0,K,S)
850 IF S=0 THEN 840
860 END
870 FOR POSITION=1 TO LEN(LABEL$)
880 LETTER$=SEG$(LABEL$,POSITION,1)
890 CODE=ASC(LETTER$)
900 CALL HCHAR(ROW,COLUMN-1+POSITION,C
    ODE)
910 NEXT POSITION
920 RETURN
```

Listing 4

```
100 REM  *************
110 REM  * THREE-BARS *
120 REM  *************
130 REM
140 REM
150 REM  ABOUT 5160 BYTES
160 REM  PRESS ANY KEY TO STOP DISPLAY
170 VERTICALMAX=200
180 SCALE=VERTICALMAX/20
190 OPTION BASE 1
200 DIM Y(3)
210 Y(1)=133
220 Y(2)=159
230 Y(3)=99.9
240 CALL SCREEN(8)
250 CALL COLOR(9,5,8)
260 CALL COLOR(10,3,8)
270 CALL COLOR(11,16,8)
280 CALL COLOR(12,2,5)
290 REM  DEFINE CHARACTERS
300 REM  FORMAT: IDENTIFICATIONS,CHARA
    CTERNUMBER,PATTERNS...
310 DATA GRID LINE,91,0000000000000000FF
    ,VERTICAL AXIS,92,0101010101010101
    ,TIC MARK,93,0101010101010101FF,BAR1
    ,96
320 DATA FFFFFFFFFFFFFFFF,BAR2,104,FFF
    FFFFFFFFFFFF,BAR3,112,FFFFFFFFFFFF
    FFFF,BASELINE,120,FF000000000000FF
330 DATA RESERVED FOR TITLE BOX
340 DATA RESERVED FOR LABELS
350 DATA RESERVED FOR LEGENDS
360 DATA RESERVED FOR ADDITIONAL CHARA
    CTERS
370 REM  DEFINE-LOOP
380 RESTORE 310
390 FOR CODE=91 TO 120
400 READ IDENTIFICATION$,CHARACTERNUMB
    ER,PATTERN$
410 IF CHARACTERNUMBER>CODE THEN 430
420 GOTO 440
430 CODE=CHARACTERNUMBER
440 CALL CHAR(CODE,PATTERN$)
450 NEXT CODE
460 REM  START SCREEN DISPLAY
470 CALL CLEAR
480 PRINT TAB(13);Y(1);TAB(18);Y(2);Y(
    3)
490 REM  ---GRAPH GRID---
500 CALL HCHAR(22,13,120,18)
510 FOR ROW=21 TO 1 STEP -1
520 CALL HCHAR(ROW,14,91,17)
530 NEXT ROW
540 LABEL$="HORSEPOWER"
550 ROW=9
560 COLUMN=1
570 GOSUB 870
580 CALL VCHAR(1,13,92,21)
590 FOR ROW=21 TO 1 STEP -5
600 ROWNUMBER=200-(10*(ROW-1))
610 LABEL$=STR$(ROWNUMBER)
620 COLUMN=10
630 GOSUB 870
640 CALL HCHAR(ROW,13,93)
650 NEXT ROW
660 REM  CALCULATE & PLOT BARS
670 MASTER$="0000000000000000FFFFFFFFFFFF
    FFFFFF"
680 FOR BAR=1 TO 3
690 BARHEIGHT=Y(BAR)/SCALE
700 YY=INT(BARHEIGHT)
710 REMAINDER=BARHEIGHT-INT(BARHEIGHT)
720 CALL VCHAR(22-YY,11+BAR*5,88+BAR*8
    ,YY)
730 CALL VCHAR(22-YY,12+BAR*5,88+BAR*8
    ,YY)
740 CALL VCHAR(22-YY,13+BAR*5,88+BAR*8
    ,YY)
750 TOPPATTERN=1+INT((REMAINDER*8)+.5)
760 STARTPOSITION=2*TOPPATTERN-1
770 TOPPATTERN$=SEG$(MASTER$,STARTPOSI
    TION,16)
780 CALL CHAR(89+BAR*8,TOPPATTERN$)
790 CALL HCHAR(21-YY,11+BAR*5,89+BAR*8
    ,3)
800 IF TOPPATTERN<>9 THEN 830
810 CALL CHAR(90+BAR*8,"0000000000000000
    FF")
820 CALL HCHAR(20-YY,11+BAR*5,90+BAR*8
    ,3)
830 NEXT BAR
840 CALL KEY(0,K,S)
850 IF S=0 THEN 840
860 END
870 FOR POSITION=1 TO LEN(LABEL$)
880 LETTER$=SEG$(LABEL$,POSITION,1)
890 CODE=ASC(LETTER$)
900 CALL HCHAR(ROW,COLUMN-1+POSITION,C
    ODE)
910 NEXT POSITION
920 RETURN
```

Listing 5

```
100 REM  ************
110 REM  * TWINKLE *
120 REM  ************
130 REM
140 REM
150 REM  ABOUT 4504 BYTES
160 REM  HOLD DOWN ANY KEY TO STOP DIS
    PLAY
170 CALL SCREEN(8)
180 CALL COLOR(9,10,8)
190 REM  SHAPED RANDOM CHARACTERS
200 RANDOMIZE
210 FOR I=1 TO 4
220 FOR I=0 TO 15
```

```
230 REM  GENERATE RANDOM NUMBERS BETWE
    EN 0 AND 15
240 N=INT((15-0+1)*RND)+0
250 REM  PUT ON CONSTRAINTS TO ELIMINA
    TE CORNERS
260 ON I+1 GOTO 280,300,340,360,410,43
    0,490,490,490,490,410,430,340,360,
    280,300
270 REM  TOP & BOTTOM ROWS
280 IF N>1 THEN 240
290 GOTO 490
300 IF N=0 THEN 490
310 IF N=8 THEN 490
320 GOTO 240
330 REM  2ND & 7TH ROWS
340 IF N>3 THEN 240
350 GOTO 490
360 IF N>12 THEN 240
370 FOURTEST=N/4-INT(N/4)
380 IF FOURTEST=0 THEN 490
390 GOTO 240
400 REM  3RD & 6TH ROWS
410 IF N>7 THEN 240
420 GOTO 490
430 IF N=15 THEN 240
440 EVENTEST=N/2-INT(N/2)
450 IF EVENTEST=0 THEN 490
460 GOTO 240
470 REM  4TH & 5TH ROWS NO CONSTRAINTS
480 REM  FOR N>9 MUST CONVERT TO HEX N
    OTATION; NOTE IN HEX NOTATION A=10
    ,B=11,C=12 ETC.
490 IF N>9 THEN 510 ELSE 630
500 ON N-9 GOTO 510,530,550,570,590,61
    0
510 GS="A"
520 GOTO 640
530 GS="B"
540 GOTO 640
550 GS="C"
560 GOTO 640
570 GS="D"
580 GOTO 640
590 GS="E"
600 GOTO 640
610 GS="F"
620 GOTO 640
630 GS=STR$(N)
640 HEX$=HEX$&GS
650 NEXT I
660 CALL CHAR(95+I,HEX$)
670 GOTO 680
680 HEX$=""
690 NEXT I
700 REM  DISPLAY TITLE
710 DATA 57,57,39,69,82,32,77,65,71,69
    ,90,73,78,69
720 CALL CLEAR
730 REM  ...BORDER...
740 FOR COL=6 TO 26
750 N=INT((8-1+1)*RND)+1
760 CALL HCHAR(14,32-COL,95+N)
770 CALL HCHAR(10,COL,95+N)
780 NEXT COL
790 FOR ROW=10 TO 14
800 N=INT(4*RND+1)
810 CALL VCHAR(ROW,6,95+N)
820 CALL VCHAR(24-ROW,26,95+N)
830 NEXT ROW
840 REM  ...TITLE...
850 CALL HCHAR(12,10,32,14)
860 RESTORE 710
870 FOR I=1 TO 14
880 READ LETTER
890 COLUMN=9+I
900 CALL HCHAR(12,COLUMN,96)
910 CALL HCHAR(12,COLUMN,LETTER)
920 NEXT I
930 REM  ...TWINKLE...
940 C=0
950 COLUMN=INT((26-6+1)*RND)+6
960 N=INT((4-1+1)*RND)+1
970 CALL HCHAR(10,COLUMN,95+N)
980 CALL HCHAR(14,32-COLUMN,95+N)
990 C=C+1
1000 IF C>45 THEN 850
1010 CALL KEY(0,K,S)
1020 IF K<>31 THEN 1040
1030 GOTO 950
1040 END
```

How to Write A BASIC Program That Writes BASIC Programs

PART 1:

A SURPRISING DISCOVERY WITH TI'S *PROGRAMMING AIDS III*

TI's *Programming Aids III* opens the door to some powerful programming techniques. The Cross Reference and Editor capabilities of this software will be appreciated by the serious Extended BASIC programmer. But the excitement really begins when you realize how this software does its thing.

PA III can provide (1) a tabular, line-number cross reference for all variables, arrays, keywords, functions, and line-number references in a program and (2) the ability to delete, move, or resequence specified groups of lines within a program much more quickly than could be done manually at the keyboard.

Required Hardware

Programming Aids III is a set of four Extended BASIC programs (LINPUT, CREF, CREFPRINT, and EDITOR) available on disk at a suggested retail price of $19.95. In addition to a disk controller, disk drive, and the Extended BASIC Command Cartridge, a printer is a practical necessity; either the TI Thermal Printer or an RS232-compatible printer may be used. In fact, there is no provision for screen display of the output from the Cross Reference procedure. (I use the inexpensive "Paper-and-Pencil Printer," however, and so modified the CREFPRINT program to display the cross reference table on the screen, using the crude SHIFT CONTINUE method to stop and start the output. These simple changes are given at the end of this chapter.)

EDITOR

The EDITOR program makes possible virtually any desired modification of line numbers in a BASIC or Extended BASIC Program. Heretofore, the only way to resequence a program was to use the RESEQUENCE (RES) command, which affects all line numbers within a program. By contrast, EDITOR allows one to resequence specified sections of a program without affecting others.

If, for instance, you have numbered subroutine statements in a manner which is easy to remember (1000, 2000, 3000, etc), you can retain this numbering and "open up" a previous part of the program for insertion of additional lines. An even more useful application would be the rearrangement of sections of BASIC code. Suppose, for example, you want to merge several programs, each of which contains subroutines. Without EDITOR, you would be faced with the time-consuming chore of moving all subroutines to the end of the merged program. With EDITOR, this procedure can be completed very simply and quickly by renumbering all subroutine lines.

Finally, the EDITOR program allows deletion of sections of BASIC code. If you want to get a subroutine out of one program to use in another, it's no problem.

How EDITOR Works

If you are wondering how a BASIC program can alter another BASIC program, be assured that it's not done with mirrors. It is a relatively simple procedure which anyone with Extended BASIC can use to write all custom utility programs and even BASIC programs which write other BASIC programs!

The technique is based upon what happens to a program when it is saved with the MERGE option (see pp.122-3 of the TI Extended BASIC manual). If you have ever cataloged a disk containing a file saved with the MERGE option, you may have noticed that, unlike an ordinary program which carries the Type description PROGRAM, a program saved with MERGE is actually a data file consisting of display code with variable length records having a maximum length of 163 bytes. A BASIC program can access this sequential file like any other file.

In addition to creating a data file form, saving a program with MERGE makes two other important changes. First, the order of program lines corresponds to the order of program line numbers. (By contrast, when a program is saved without MERGE, the file is a program memory image, and lines are placed in program memory in the order in which they were entered—not according to line number.) Second, the content of each line is represented in condensed format: All non-essential information is deleted in a coding process. When a program saved with the MERGE option is loaded into memory with the MERGE Command and LISTed (see *TI Extended BASIC* Manual, page 114), the coding process is reversed and each program is reconstructed.

In order to understand how the EDITOR program works, it is necessary to know how line numbers are represented in condensed format. The first two bytes of each record contain the line number represented in ASCII code. Table 1 shows how the line numbers "80" and "9020" are represented in ASCII characters. Starting with the line

number 80, the first step involves representing the base 10 number in binary. Two bytes (8 bits each) are available for this representation. Next, the base 10 representation of each byte is determined and the corresponding ASCII symbol produced. In this case, the character with an ASCII code of 80 is "P". Applying this process to the number 9020 gives the ASCII representation "#<".

Table 1
ASCII Coding of Line Numbers

Line Number	80	
	Byte 1	Byte 2
Binary	00000000	01010000
Base 10	0	80
ASCII		P

Line Number	9020	
	Byte 1	Byte 2
Binary	00100011	00111100
Base 10	35	60
ASCII	#	<

Table 2
Sample Cross Reference Output

```
MUSIC 2/1
PROGRAM UNIT (MAIN)
STRING ARRAYS
N$ ( )
   100
   120
   140
NUMERIC ARRAYS
NT ( )
   100
   120
   340
NUMERIC VARIABLES
I
   110
   120
   140
   165
   240
J
   150
   170
BASIC KEYWORDS
CALL
   130
   230
   240
   250
DATA
   190
   200
DIM
   100
FOR
   110
   150
GOSUB *
   160
NEXT
   170
   180
PRINT
   140
READ
   120
REM
   220
RETURN
   250
STOP
   210
BASIC FUNCTIONS
&
   140
SUBPROGRAMS
CLEAR
   130
SOUND
   230
   240
   250
LINE REFERENCES
   220
   180
                    ...
```

In condensed code format, when the left-most bit of a byte is "on," the software which reconstructs a program from the code is signaled that some special action will be required in the reconstruction process. In the case of line numbers, this principle applies to the first bit of the first of the two line-number bytes. When all bits except the left-most one are "on" in both bytes, the number represented in base 10 is 32767 (in binary, 01111111 11111111), the highest allowable line number in a program. When the left-most bit is added, the two-byte combination becomes an end-of-file mark. Thus the first two bytes of the last condensed format record must be CHR$(255)&CHR$(255), equivalent to 65535 in base 10.

With this information, you should be able to understand the basic operation of the EDITOR program. The program to be edited is first saved with the MERGE option, and then the EDITOR program is loaded and run. Upon entry of the "OLD" command provided, EDITOR inputs each record in the condensed format file and constructs the line number from the ASCII codes of the first two bytes. Program line numbers thus obtained are stored in an array, with array position corresponding to record number. After the user has altered these numbers using the DELETE (DEL) and RESEQUENCE (RES) commands provided, the SAVE command initiates the process in which altered numbers are reassigned to records in the file. As each record is read a second time, the corresponding line number in the array is translated into two ASCII characters which are substituted for those on the record, and the new record is written to a new file (after making the necessary changes to any line references). At the end of this process, the end-of-file mark is written as the last record on the new file. After initializing program memory with the NEW command, all you need to do is load the new file with the MERGE command. The program will then be reconstructed and can be SAVEd in the usual way.

CROSS REFERENCE

The remaining three programs (LINPUT, CREF, and CREFPRINT) are used to produce a complete tabulation of all lines in which each variable, array keyword, function, and line number reference occurs. An independent tabulation is provided for each subprogram. The cross reference table will give you detailed documentation for use in program development, and would also seem to be a useful tool in analyzing a poorly documented program. (See Table 2)

As in the case of the EDITOR program, the first step involves saving the program to be cross referenced by using the MERGE option. The LINPUT program converts the DISPLAY records of the merged file to INTERNAL code, presumably to speed subsequent execution. The CREF program then reads in each record of the file and analyzes its contents for the presence of all keywords, functions, etc., which occur in TI Extended BASIC, as well as in the user's variable names, arrays, line references, and subprograms. The output, a list of the line numbers in which each element is found, is written to a disk file. The file is then printed by the CREFPRINT program.

The instructions recommend that the CREF program be run in TI BASIC, rather than Extended BASIC, to speed execution. Even with this advantage, however, the cross referencing of a large program should be planned so that you can be doing something else—like taking a trip to Switzerland. Actually, it doesn't take quite that long: Cross-referencing a program of moderate size (270 lines) takes 35 minutes.

HOW CREF Works.

Although a detailed analysis of the cross reference program is beyond the scope of this article, generalization the principles involved presumes an understanding of the structure of condensed code. As mentioned previously, the method used to signal the reconstruction software that it is encountering an "instruction" byte involves an "on" condition in the left-most bit. In contrast to line numbers, most "instructions" in condensed code consist of a single byte. When the left-most bit is "on" (i.e., 10000000) the base 10 representation is 128. Instructions thus begin with the number 10000001 or 129

ASCII byte codes used by the reconstruction software to generate BASIC keywords, punctuation, etc., are translatable with the program *Condensed Format Code Table*. This program generates a file called DSK1.FILENAME which is in condensed format. Each record in the file contains a single byte in the third position beginning with ASCII 129 and ending with ASCII 254. This byte will be interpreted as an "instruction" by the reconstruction software. Preceeding the byte, a two-byte line number is written; following it is an end-of-line mark, ASCII 0. Line numbers have been set equal to the ASCII code for ease in subsequent interpretation of the results.

In order to view the reconstruction of each potential BASIC element, you first initialize program memory with

NEW command, then load the output file with the RGE command, as if it were a program, i.e., MERGE K1.FILENAME. The result is given in Table 3. For le:

HR$(129) is reconstructed as ELSE
HR$(130) as : :
HR$(166) as WARNING

```
100 REM  ********************
110 REM  *                  *
120 REM  *  CONDENSED FORMAT *
130 REM  *     CODE TABLE   *
140 REM  *                  *
150 REM  ********************
160 REM
170 REM
180 REM
190 REM
200 REM  ********************
210 REM
220 REM    OPEN OUTPUT FILE
230 REM    'DSK1.FILENAME'
240 REM    USING PARAMETERS OF
250 REM    MERGED FILE FORMAT
260 REM
270 REM  ********************
280 REM
290 OPEN #1:"DSK1.FILENAME",DISPLAY ,O
    UTPUT,VARIABLE 163
300 REM
310 REM  ********************
320 REM
330 REM   BEGIN COUNTING (I)
340 REM   WITH HIGH BIT ON
350 REM    I.E.  129 130 ... 254
360 REM
370 REM  ********************
380 REM
390 FOR I=129 TO 254
400 REM
410 REM  ********************
420 REM
430 REM   CALCULATE VALUES FOR
440 REM   FIRST TWO BYTES TO
450 REM   REPRESENT LINE NO.S
460 REM   SO THAT LINE NO WILL
470 REM   EQUAL ASCII CODE
480 REM
490 REM  ********************
500 REM
    LNBYTE1=INT(I/256)
    LNBYTE2=I-256*LNBYTE1
530 REM
540 REM  ********************
550 REM
560 REM   WRITE RECORD:
570 REM
580 REM   BYTE#1&2=LINE NUMBER
590 REM   BYTE#3=CODED BASIC
600 REM   BYTE#4=END OF LINE
610 REM
620 REM  ********************
630 REM
640 PRINT #1:CHR$(LNBYTE1)&CHR$(LNBYTE
    2)&CHR$(I)&CHR$(0)
650 REM
660 REM  ********************
670 REM
680 REM   REPEAT LOOP FOR NEXT
690 REM   ASCII CODE.
700 REM
710 REM  ********************
720 REM
```

```
730 NEXT I
740 REM
750 REM  ********************
760 REM
770 REM   WRITE END OF FILE
780 REM   MARK = LINE NUMBER
790 REM   OF 65535
800 REM
810 REM  ********************
820 REM
830 PRINT #1:CHR$(255)&CHR$(255)
840 REM
850 REM  ********************
860 REM
870 REM   CLOSE FILE AND STOP
880 REM
890 REM  ********************
900 REM
910 CLOSE #1
920 STOP
```

Table 3
Condensed Format Code Table

128 ELSE	171 ???	213 LEN
129 : :	172 ???	214 CHR$
130 !	173 ???	215 RND
131 IF	174 ???	216 SEG$
132 GO	175 ???	217 POS
133 GOTO	176 THEN	218 VAL
134 GOSUB	177 TO	219 STR$
135 RETURN	178 STEP	220 ASC
136 DEF	179 !	221 PI
137 DIM	180 :	222 REC
138 END	181 :	223 MAX
139 FOR	182)	224 MIN
140 NEXT	183 (	225 RPT$
141 LET	184 &	226 ???
142 BREAK	185 ???	227 ???
143 UNBREAK	186 OR	228 ???
144 TRACE	187 AND	229 ???
145 UNTRACE	188 XOR	230 ???
146 INPUT	189 NOT	231 ???
147 DATA	190 =	232 NUMERIC
148 RESTORE	191 <	233 DIGIT
149 RANDOMIZE	192 >	234 UALPHA
150 NEXT	193 +	235 SIZE
151 READ	194 -	236 ALL
152 STOP	195 *	237 USING
153 DELETE	196 /	238 BEEP
154 REM	197 ^	239 ERASE
155 ON	198 ???	240 AT
156 PRINT	199 ???	241 BASE
157 CALL	200 ???	242 ???
158 OPTION	201 ???	243 VARIABLE
159 OPEN	202 EOF	244 RELATIVE
160 CLOSE	203 ABS	245 INTERNAL
161 SUB	204 ATN	246 SEQUENTIAL
162 DISPLAY	205 COS	247 OUTPUT
163 IMAGE	206 EXP	248 UPDATE
164 ACCEPT	207 INT	249 APPEND
165 ERROR	208 LOG	250 FIXED
166 WARNING	209 SGN	251 PERMANENT
167 SUBEXIT	210 SIN	252 TAB
168 SUBEND	211 SQR	253 @ (TAB)
169 RUN	212 TAN	254 VALIDATE

Table 4
Condensed Record Structure

OPEN #1:"DSK1.BASIC",INPUT,DISPLAY,VARIABLE 163

ASCII CODE FOR LINE 100

3	159*	23	179*
4	253*	24	182*
5	200*	25	178*
6	1	26	243*
7	49 1	27	200*
8	181*	28	2
9	199*	29	49 1
10	10	30	54 6
11	68 D	31	51 3
12	83 S	32	0
13	75 K		
14	49 1		
15	46 .		
16	66 B		
17	65 A		
18	83 S		
19	73 I		
20	67 C		
21	179*		
22	146*		

PRESS ANY KEY TO CONTINUE

At the same time, several codes are reconstructed into things which can't be understood directly (e.g., 171-175, 185, and 198-201). It is apparent that some of the ASCII codes are used for purposes other than direct translation to BASIC. Some might be used as descriptors of subsequent bytes (e.g., for purposes of identifying trailing bytes as

numeric data, line numbers references, string data, etc.) while other of these ASCII codes may not be assigned at all.

Putting this question aside for the moment, let us see how we could write a program that would remove all REM statements from another program. The ASCII code for REMARK (REM) is found in Table 3 to be 154. If we assume that the ASCII character with code 154 will be found in the third position of a REM statement in condensed format (following the line-number bytes), we can write a *REM Remover* program very simply. Such a program would need to read a record from a program file saved with the MERGE option, see if the third byte is CHR$(154), and if not, print the record in a second file. That is what the following program does. To use it with the "Condensed Format Code Table" program, save that program with the MERGE option (SAVE DSK1.CODE,MERGE), run the *REM Remover*, and load the output file, DSK1.REMFREE, with the MERGE command (MERGE DSK1.REMFREE). Presto, Chango! LISTing the program shows it to be "REMless," and this version may now be saved in the usual way under a new file name.

```
100 REM ********************
110 REM *    REM REMOVER     *
120 REM ********************
130 REM
140 REM
150 REM
160 REM
170 PRINT "ENTER FILE NAME"
180 INPUT "DSK1.XYZ-":XS
190 OPEN #1:XS,DISPLAY ,INPUT ,VARIABL
    E 163
200 OPEN #2:"DSK1.REMFREE",DISPLAY,OU
    TPUT,VARIABLE 163
210 EOFS=CHR$(255)&CHR$(255)
220 LINPUT #1:XS
230 IF SEG$(XS,1,2)=EOFS THEN 270
240 IF SEG$(XS,3,1)=CHR$(154)THEN 260
250 PRINT #2:XS
260 GOTO 220
270 PRINT #2:CHR$(255)&CHR$(255)
280 CLOSE #1
290 CLOSE #2
300 STOP
```

Of course, more complex applications require a more detailed knowledge of condensed format structure. The *Condensed Record Structure* program listed below will allow you to examine the condensed structure of every line in any BASIC program. With such a representation and the list of codes in Table 3, a great deal of additional information can be deduced.

For purposes of illustration, let us treat the "Record Structure" program itself as the program to be analyzed. First enter the program without the REM statements, and then save it as DSK1.BASIC,MERGE. Now enter RUN to display the code structure of each line. The display for the first line is shown in Table 4.

The first column in each pair of columns shows the position of the byte code. The first position displayed is 3 because 1 and 2 are used for the line number. An asterisk has been placed beside all ASCII codes which exceed 128 to easily identify them as "instruction" codes. Codes which are between 32 and 94 are followed by their corresponding ASCII character representations.

```
100 REM ********************
110 REM *                  *
120 REM *  CONDENSED RECORD  *
130 REM *      STRUCTURE     *
140 REM *                  *
150 REM ********************
160 REM
170 REM
180 REM
190 REM
200 OPEN #1:"DSK1.BASIC",INPUT ,DISPLA
    Y ,VARIABLE 163
210 LINPUT #1:XS
220 BYTE1=ASC(SEG$(XS,1,1))
230 BYTE2=ASC(SEG$(XS,2,1))
240 LINENUM=BYTE1*256+BYTE2
250 IF LINENUM=65535 THEN 430
260 DISPLAY AT(1,3)ERASE ALL:"ASCII CO
    DE FOR LINE "&STR$(LINENUM)
270 COL=1 :: J=0
280 FOR I=3 TO LEN(XS)
290 IF I>62 THEN 400
300 ROW=I-2*(COL-1)
310 J=J+1
320 DISPLAY AT(ROW,COL):STR$(J)
330 Y=ASC(SEG$(XS,I,1))
340 DISPLAY AT(ROW,COL+3):STR$(Y)
350 IF Y>128 THEN DISPLAY AT(ROW,COL+5
    ):"*"
360 IF Y>31 AND Y<91 THEN DISPLAY AT(R
    OW,COL+6):CHR$(Y)
370 IF I<20 THEN 390
380 COL=COL+10 :: J=-1
390 NEXT I
400 DISPLAY AT(24,2)BEEP:"PRESS ANY KE
    Y TO CONTINUE"
410 CALL KEY(0,K,S):: IF S=0 THEN 410
420 GOTO 210
430 STOP
```

Since it is known that the first line of the program is OPEN #1;"DSK1.BASIC", INPUT, DISPLAY, VARIABLE 163, let us see what sense can be made of the corresponding condensed code. Codes 159 and 253 correspond to OPEN and #. Although the meaning of code 200 is not known, in looking ahead to colums 6 and 7 we might hypothesize that 200 means "A number is about to be encountered, and the next byte will give the number of bytes used to represent that number."

Although 181 is a ":", 199 is another unknown. Looking ahead at positions 10-20, we might again hypothesize that 199 is used for strings, in the way that 200 is used for numbers. The "10" in position 10 is consistent with this hypothesis since DSK1.BASIC is 10 characters long. Next, we encountered the codes for INPUT, DISPLAY, VARIABLE. In position 27 another 200 is encountered, and the hypothesis applied earlier to the 200 in position 5 is consistent with what follows—a "3" in position 28 followed by the 3 numbers "163". Finally, a 0 is encountered that indicates end-of-line. By writing program lines specifically for the purpose, you can use the *Condensed Record Structure* program to deduce additional information about condensed format.

How to Write A BASIC Program That Writes BASIC Programs

PART 2:

RULES OF MERGE FORMAT

In the previous section, MERGE format was discussed in connection with TI's *Programming Aids III*. When an Extended BASIC program is saved with the MERGE option (disk only), a data file is written such that each record in the file contains a coded representation of one line of BASIC code. This file can then be loaded into program memory with the MERGE command. Because the file is a data file, it can also be generated by a BASIC program. If all of the rules of MERGE format are observed, the file is indistinguishable from one created by saving a program with the MERGE option and can be loaded into program memory with the MERGE command. Thus an Extended BASIC program can, in effect, write another Extended BASIC program.

One can think of a variety of contexts in which this program generation capability could be used. For instance, a program might allow preparation of music or graphics in an interactive, "high level" format and then use this data to write a BASIC program or subroutine which produces the music or graphics display.

File Structure

The MERGE format file consists of sequentially organized records, each corresponding to one line of BASIC code. Records are of variable length with a maximum length of 163 display format characters. The OPEN statement for a MERGE format file might be:

```
OPEN #1:"DSK1.FILENAME",VARIABLE 163
```

Record Structure

Records in the file each represent a line of BASIC as strings of ASCII characters. The ASCII codes of the first two characters represent the line number, the last character designates "end-of-line", and the BASIC statement(s) are represented in coded form in between.

Let's consider first how line numbers are represented. You are probably aware that code numbers are associated with the character patterns used to display information. The character associated with a code can be obtained with the CHR$ function; PRINT CHR$(65) displays the pattern of the character with ASCII code 65 on the screen—the letter A. (ASCII, by the way, stands for American Standard Code for Information Interchange.)

While some ASCII characters, like the letter A, have an associated pattern, others do not. However, any of the 256 ASCII characters can be accessed with the CHR$ function and subsequently used in strings just like any of the more familiar characters. PRINT CHR$(32)&CHR$(255) displays two characters. Neither has a pattern, so neither can be seen, but the computer is able to recognize each character nevertheless.

A character consists of a "byte," and a byte can be thought of as an eight-place binary number. Just as the decimal number system contains 10 digits (0-9), the binary system contains two digits, 0 and 1. In the decimal system, the first place to the left of the decimal point counts in units of one. Each successive place counts in units of the number base 10 multiplied times the units of the preceeding place—i.e.,1's, 10's, 100's, 1000's, etc. Similarly the first place in a binary number counts 1 and successive places in units of the number base 2, multiplied times the units of the preceeding base; i.e., 1's, 2's, 4's, 8's, 16's, etc. Thus the eight-place binary number 00110001 is equivalent to 0+0+32+16+0+0+0+1 or 49 in decimal. The binary number 11111111 is equivalent to 255 (128+64+32+16+8+4+2+1), and this is the largest ASCII code because it is the largest number that can be represented with a byte. The 256 ASCII characters are thus numbered from 0 through 255.

The decimal equivalent of ASCII code is used to represent the line number, but with only one byte, the largest line number which could be represented is 255. To allow representation of high line numbers, a second character is added giving a total of 16 binary places. Applying the same principle used above, the places count (from right to left) in units of 256 (128*2), 512, 1024, etc. When placed in the first two positions of a MERGE format record, CHR$(2)&CHR$(8)—i.e., 00000010 00001000—would represent the line number 520 (512+8). A quick method of determining the decimal representation of any two characters is to multiply the code of the first by 256 and add the code of the second. In the above example, 520=2*256+8.

The highest allowable line number in TI BASIC is 32767 (01111111 11111111 or CHR$(127)&CHR$(255)). Adding the left digit gives the end-of-file mark used in MERGE format, equivalent to a line number of 65535. These two bytes, CHR$(255)&CHR$(255) must be in the first two positions of the last record in a MERGE format file.

Just as these two characters signal the end of the file, the byte CHR$(0) is used to signal the end of each line. This character must be the last one in each record.

MERGE Format Code

This brings us to the question of what to put between the line number and end-of-line mark and before the end-of-file mark, viz., the coded BASIC statements. Many elements which comprise Extended BASIC statements are listed in Table 3 together with their ASCII character tokens. In MERGE format, the BASIC elements listed are represented by a single ASCII character. For instance, CHR$(156) represents PRINT, CHR$(130) the statement separator, CHR$(213) the LEN function, etc. In order to prepare BASIC statements in MERGE format, however, one must also know how to represent variable names, numeric and string constants, and line numbers occurring within statements.

The easiest of these to represent is the variable name; the normal ASCII representation for each character of the name is used. Consider the line:

```
10 PRINT XYZ
```

The MERGE format record used to represent this line would be:

```
CHR$(0)&CHR$(10)&CHR$(156)&"XYZ"&CHR$(0)
```

That is, seven bytes would be concatenated in a string and written in the appropriate disk file record. The first two bytes represent the line number; the next, the keyword PRINT; the next three, the variable name; and the last, the end-of-line mark. Assuming that the complete file corresponds to the requirements of MERGE format in other respects, when loaded into program memory with the MERGE command LISTing, the program will show it to contain the line intended.

Numeric constants and unquoted string constants are handled differently from variable names: Each number of unquoted string must be preceded by two identifying bytes. The first is CHR$(200), the character which signals the beginning of an unquoted string. Following CHR$(200), a byte must be included to indicate the number of subsequent characters in the string or number. This byte is simply the character with the code equal to the length of the string—i.e., if the string were five characters long, CHR$(5) must be included; if 12 characters, CHR$(12). For example, consider the statement,

```
10 PRINT X+345
```

The statement would be represented in MERGE format with 11 bytes as follows:

```
CHR$(0)&CHR$(10)&CHR$(156)&"X"&CHR$(193)
&CHR$(200)&CHR$(3)&"345"&CHR$(0)
```

Here, CHR$(200)&CHR$(3)&"345" first indicates that an unquoted string is to be encountered, then indicates how long that string is, and finally gives the string.

Quoted strings are handled in much the same way, except that CHR$(199) is used instead of CHR$(200):

```
10 RUN "DSK.1.FILENAME"
```

would be represented as

```
CHR$(0)&CHR$(10)&CHR$(169)&CHR$(199)&
CHR$(13)&"DSK1.FILENAME"&CHR$(0)
```

Notice that quote marks are *not* explicitly included in the string representation. They are automatically provided for by the use of CHR$(199).

Finally, line numbers included in program statements such as GOTO and GOSUB must consist of two bytes coded in the same way as the line number bytes which begin each record. Moreover, these two bytes must be preceded by CHR$(201) to indicate that they are to be interpreted as a line number. The statement:

```
10 GOTO 200
```

would be represented as follows:

```
CHR$(0)&CHR$(10)&CHR$(134)&CHR$(201)
&CHR$(0)&CHR$(200)&CHR$(0)
```

Program Generation

Although MERGE format programs can be generated with the above technique, its use would be cumbersome—to say the least. The following method simplifies the process considerably.

For the moment, let's put aside the question of generating the portion of the character string associated with the BASIC statement. Assume that this string is generated and assigned to the string variable LINE$. Each time a LINE$ string is constructed, two line number bytes must be added to the beginning, an end-of-line byte to the end, and the whole thing must then be written as a record in the MERGE format file. The easiest way to handle the operations which follow the construction of LINE$ is to use a subroutine. Given a starting line number, LN, the following subroutine constructs the two-byte ASCII line number representation and writes the file record. It then increments the line number by 10.

```
9000 PRINT #1: CHR$(INT(LN/256))&CHR$
(LN-256*INT(LN/256))&LINE$&CHR$(0) ::
LN=LN+10 :: RETURN
```

After the BASIC statement portion of the record is assigned to LINE$, a simple GOSUB 9000 takes care of all the rest.

The construction of LINE$ strings can be simplified by assigning ASCII character codes to string variables with easy to remember names. For instance:

```
100 REM$=CHR$(154)::FOR$=CHR$(140)::NEXT$
=CHR$(150)::IF$=CHR$(132)::THEN$=CHR$(176)
::TO$=CHR$(177)
```

Some string functions are followed by a "$" and are reserved words. But in TI BASIC, they can be embedded in a variable name so that one could use variable names like @SEG$, @STR$, etc., for storage of the appropriate ASCII character. Punctuation, arithmetic operators, and characters 199-201 also must be assigned "creative" string variable names: Q$ for quoted string, UQ$ for unquoted string, CM$ for comma, etc.—whatever will be easiest for you to remember.

The next level of simplification involves user-defined functions to include more than one byte whenever possible. For example, it is clear that CALL will always be followed by an unquoted string; CALL COLOR, CALL SPRITE, CALL SOUND, etc. For that matter, the unquoted string token will always be followed by a byte indicating string length. Construction of strings which in-

clude the call keyword can therefore be simplified by defining function appropriately:

```
110 DEF UQ$(X)=CHR$(200)&CHR(X)::CALL$(X)
=CHR$(157)&UQ$(X)
```

atement like CALL SCREEN (2) can then be written:

```
120 LINE$=CALL$(6)&"SCREEN"&LP$&UQ$(1)&
"2"&RP$::GOSUB 9000
```

(if CHR$(183), the left parenthesis, had previously been assigned to LP$ and 182, the right parenthesis, to RP$)

By making the function definitions a little more complex, the statement can be even further simplified:

```
110 DEF UQ$(X$)=CHR$(200)&CHR$(LEN(X$))&X$
120 DEF CALL$(X$)=CHR$(157)&UQ$(X$)
```

makes it possible to write CALL SCREEN (2) like this:

```
130 LINE$=CALL$("SCREEN")&LP$&UQ$("2")&
RP$::GOSUB 9000
```

t's beginning to look a lot like BASIC.

Built-in functions can similarly be defined to facilitate onstrucion of MERGE format strings. For instance,

```
40 DEF INT$(X$)=CHR$(207)&LP$&X$&RP$
```

llo one to write X=INT(Y/256) as

```
50 LINE$="X"&EQ$&INT$("Y"&DIV$&UQ$
("256"))::GOSUB 9000
```

f CHR$(190) had been previously assigned to EQ$ and HR$(196) TO DIV$)

Similarly, line numbers occurring within statements, ch as GOTO or GOSUB, can be simplified with the llowing function:

```
0 DEF LN$(X)=CHR$(201)&CHR$(INT(LN/256))
CHR$(LN-256*INT(LN/256))
```

that the statement GOTO 200 can be written simply as

```
70 LINE$=GOTO$&LN$(200) :: GOSUB 9000
(GOTO$ had previously been assigned CHR$(134))
```

Using string variable names and user-defined string nctions, you can create your own custom "language" use in writing MERGE format records.

The following program may help to tie up the concepts sented; it is a trivial example of a music program ie tor. The program writes CALL SOUND tements in the MERGE format file "DSK1.BASIC" the user presses a single key.

```
100 REM *MUSIC PROGRAM GENERATOR*
110 REM
120 REM
130 REM
140 REM ASSIGN STRING VARS
150 REM
160 CM$=CHR$(179)::  LP$=CHR$(183)::  RP
$=CHR$(182)
170 REM
180 REM DEFINE STRING FUNCTIONS
190 REM
200 DEF UQ$(X$)=CHR$(200)&CHR$(LEN(X$)
)&X$  ::  DEF CALL$(X$)=CHR$(157)&UQ
$(X$)
210 REM
220 REM ASSIGN FIRST LINENO
230 REM
240 LN=100
250 REM
260 REM OPEN MERGE FILE
270 REM
280 OPEN #1:"DSK1.BASIC",VARIABLE 163
290 REM
300 REM DISPLAY INSTRUCTIONS
310 REM
320 DISPLAY AT(7,1)ERASE ALL:"TO ENTEN
 A NOTE PRESS ONE OF THE FOLLOWING
KEYS:-"
330 DISPLAY AT(12,5):"A B C D E F G":
 :: DISPLAY AT(20,7):"PRESS 7 TO STO
P"
340 REM
350 REM ACCEPT KEY INPUT
360 REM
370 CALL KEY(0,KEY,STATUS)::  IF KEY=80
 THEN 500 ELSE IF KEY<65 OR KEY>71
 THEN 370
380 KEY=2*KEY-130  ::  IF KEY>2 THEN KEY
=KEY-1  ::  IF KEY>7 THEN KEY=KEY-1
390 REM
400 REM COMPUTE FREQUENCY
410 REM AND PLAY NOTE
420 REM
430 FREQ=INT(440*(2^(1/12))^KEY+.5)::
CALL SOUND(500,FREQ,0)
440 REM
450 REM FORM MERGE STATMENT
460 REM
470 LINE$=CALL$("SOUND")&LP$&UQ$("500"
)&CM$&UQ$(STR$(FREQ))&CM$&UQ$("0")
&RP$  ::  GOSUB 490
480 GOTO 350
490 PRINT #1:CHR$(INT(LN/256))&CHR$(LN
-256*INT(LN/256))&LINE$&CHR$(0)::
LN=LN+10  ::  RETURN
500 REM WRITE END OF FILE
510 REM
520 PRINT #1:CHR$(255)&CHR$(255)
530 CLOSE #1
540 REM
550 REM LOAD INSTRUCTIONS
560 REM
570 DISPLAY AT(12,1)ERASE ALL:"TO LOAD
 THE PROGRAM:-"  ::  DISPLAY AT(15,2)
:"1- ENTER 'NEW'"
580 DISPLAY AT(17,2):"2> ENTER 'MERGE
DSK1.BASIC'"  ::  DISPLAY AT(19,2):"
3> ENTER 'RUN'"  ::  END
```

HOW EXTENDED IS EXTENDED BASIC?

Nothing caused as much excitement and anticipation in the TI-99/4A community as the announcement (which now seems like an eternity ago) that Extended BASIC would be forthcoming. Well, now that the new programming language is being gobbled up by hungry Home Computer users, the question on everyone's mind is, naturally enough, "Was it worth waiting for?"

For the answer to this, and to help put the new software in proper perspective, we should first examine TI's claims for the language (in the introduction to the reference manual): "Texas Instruments Extended BASIC. . .has the features expected from a high level language plus additional features not available in many other languages, including those designed for use with large, expensive computers." The key words here are "expected" and "not available." Features such as DISPLAY AT, ACCEPT AT, PRINT. . .USING, IMAGE, ON ERROR, multiple statement lines, expanded IF-THEN-ELSE statements, PEEK, Boolean operators, and assembly language subroutine calls are indeed "expected." Unfortunately, they were expected in the *ordinary* TI BASIC, since they're *standard* features of various Microsoft BASICs found in other machines. But just as plain, old, ordinary TI BASIC has its share of surprises that *aren't* commonly found in other BASICs (e.g., CALL SAY, RESEQUENCE, *complete* EDIT, TRACE, and BREAK utilities, plus its marvelously simple character definition and color assignment facilities), TI Extended BASIC also has its own unique bag of tricks *not found* on other machines. And this bag of tricks includes some mighty impressive feats of computing magic.

But before we get into these *extended* features, let's examine some of the obvious changes from TI BASIC. First, there's the matter of a slight reduction in usable RAM. The maximum program size in Extended BASIC is 864 bytes smaller than in TI BASIC. Although this represents only about a 6% reduction, *any* reduction in user memory is significant if it prevents certain applications from being RUN. And, in fact, as little as 500 bytes is frequently the critical amount of extra memory needed. (Witness the several programs in this volume that cannot be loaded or RUN with the disk controller's power on—even with the CALL FILES(1) command that frees all but the 500 bytes for the disk system.) So programmers without the 32K RAM expansion should try wherever possible to make up the loss with Extended BASIC's built-in memory saving features: multiple statement lines (with more allowable characters per line), expanded IF-THEN-ELSE statements, multiple variable assignments, trailer comments that immediately follow statements (instead of separate REMs), repetition of strings with the RPT$ function, and the use of MIN and MAX functions.

The loss of user-definable characters in the character sets 15 and 16 is another departure from the TI BASIC standard. These custom characters are no longer available to programmers since the memory area is needed to keep track of sprites. Therefore, a TI BASIC program that doesn't use these character sets is supposed to RUN in Extended BASIC in most circumstances—unless, of course, you've done something that will obviously cause trouble, such as accidentally using a TI Extended BASIC keyword as a variable in your TI BASIC program (e.g., DIGIT, ERASE, ERROR, IMAGE, MERGE, MAX, MIN, SIZE, WARNING, etc.) [See the July/August 1981 issue of *99'er Magazine* for an analysis of what *is* and *isn't* interchangeable.—Ed.]

Now, let's take a peek (no pun intended) into the "bag of tricks" I mentioned earlier. A good place to start is with Extended BASIC's exciting new graphics capabilities. Nine new subprograms (plus 2 redesigned ones) provide the ability to create and thoroughly control the shape, color, and motion of smoothly-moving high-resolution graphics. These are the true *sprites*—graphics that can be displayed and moved at any of 49,152 positions (192 rows × 256 columns) rather than the 768 positions (24 rows × 32 columns) CALLed by the VCHAR and HCHAR statements of TI BASIC. But that's only the beginning. Sprites can be set in motion with simple X and Y velocity components and will continue their motion without further control; they can grow and shrink at will, be relocated or "hidden", and even pass over and blot out fixed objects and other sprites to give the illusion of depth and 3-D animation. [This is a function of the three-dozen stacked image planes of the Home Computer's video display processor chip—a unique graphics display explained more fully in "3-D Animation."—Ed.]

Although game aficionados and educators have every right to be overjoyed with the new sprites capability, TI-99/4A users who are more interested in business, scientific and professional applications will be drawn to other Extended BASIC features. First on the list is the impressive subprogram capability. Several options exist for passing values (and entire arrays) between main and

subprograms. There's also built-in protection to prevent a subprogram's *local* variables from affecting the *main* variables. Additionally, commonly used subprograms would be SAVEd on a separate disk, and later MERGEd. This will allow programmers to build up a library of "universal" subprograms that can be called upon to supply the appropriate cartridges for new programming tasks—without time-consuming re-coding and debugging.

If this new subprogram flexibility is not enough for your most demanding tasks, how about "program chaining," where one program can load and RUN another program from a disk. This means that multi-part programs of almost *unlimited* size can now RUN on the TI-99/4A if they are broken into pieces and each segment is allowed to RUN the next. And at any point in this chain, a "menu" may be inserted, allowing the user to choose with a *single keystroke* the particular program to be RUN. Imagine the possiblities!

Those of you with a speech synthesizer, or thinking of purchasing one, will be happy to learn that Extended BASIC includes a speech editor. You will no longer need the separate Command Cartridge (with a retail price of about $45). What's more, with the combination of CALL SPGET, the capability of subroutine MERGEs, and the salvaging of the code patterns (that TI supplies in the appendix of the reference manual), you can now easily add the suffixes ING, S, and ED to the roots of words in the resident vocabulary. And if TI ever supplies users with their master file of coded speech patterns and rules for combining them, it will be possible to create your own new words. As of now, TI provides only one cryptic statement: "Because making new words is a complex process, it is not discussed in this manual."

Incidentally, this capability of having the computer say what you want it to say rather than being limited to a fixed vocabulary will, in fact, be implemented through a related approach. I'm referring to the "text-to-speech" capability of the forthcoming *Terminal Emulator II* Command Cartridge, which is programmable in TI BASIC. Since only one Command Cartridge at a time can be attached to the TI-99/4A, text-to-speech cannot be used with the *Extended BASIC* Command Cartridge. [See "Text to Speech on the Home Computer."—Ed.]

The final two features I'm going to cover in this overview provide a fair degree of software protection and open the door to additional language capabilities. Consequently, these are the particular features that may have the most profound impact on the entire TI-99/4A community—ultimately determining the quality and quantity of most of the commercial software for this machine.

Extended BASIC programs can be SAVEd in a PROTECTed form to guard against software piracy. This irreversible feature allows a program to be RUN or loaded into memory only with an OLD command. A program thus PROTECTed cannot be LISTed, EDITed, or SAVEd. If the program was originally SAVEd and PROTECTed on a disk, you must still use the protect feature of the *Disk Manager* Command Cartridge to completely "lock up" the software by preventing it from being copied as well.

Extended BASIC has the capability to CALL and RUN assembly language programs if the 32K RAM expansion peripheral is attached to the computer. Since Assembly Language has a much faster execution speed than BASIC, many applications programs that are unfeasible to write in either TI BASIC or TI Extended BASIC (and Extended BASIC is not significantly faster than its predecessor) can now be written in TMS9900 Assembly Language, LOADed into the expansion memory peripheral, and RUN on a TI-99/4A. This paves the way for some fairly sophisticated applications programs that can now be targeted for TI-99/4A users. [See the related assembly language sections in this book.—Ed.]

Even though a TI-99/4A with Extended BASIC and the memory expansion peripheral can CALL and RUN Assembly Language programs and subroutines, it cannot be used to *write* them *at present*. And instead of a direct implementation of the POKE command, TI gave users an indirect implementation. To load data directly into memory locations, they can use CALL LOAD with the optional fields specifying a starting address followed by data bytes. The TMS9900 Assembler, available on the *Editor/Assembler* Command Cartridge and its accompanying diskettes, allows Home Computer owners to write their own Assembly Language programs and call them up through Extended BASIC. Besides this obvious use of an assembler, it opens up other exciting possibilities: More exotic languages can be written in TMS9900 Assembly Language especially for TI-99/4A implementations. FORTH, for instance, is now available.

The bottom line is more software tools for developers and more economic incentive for them to produce valuable programs that can be protected against most piracy. This means that the TI-99/4A user community will be seeing a lot more useful software enter the market. Being able to run this software should more than justify the $100 (retail) price for this filled-to-capacity 36K byte TI *Extended BASIC* Command Cartridge with accompanying 224-page reference manual. Therefore, the answer to the title's rhetorical question, "How Extended is Extended BASIC?" is apparently, "Extended enough. . . ."

POCKET TOWER OF HANOI

You are in an ancient temple at the center of the earth where three diamond needles bear eighty golden rings of graduated sizes. At the beginning of time the rings were all on one needle; but now the temple monks are transferring the rings, one at a time, from needle to needle, never setting a ring on a smaller ring. When they have moved all eighty rings to one of the other two needles, the world will end . . .

Possibly you have seen a children's toy along these lines—four or five disks of various colors and sizes, drilled to fit on three wooden pegs. The object is to start with the disks on one peg, and by moving one at a time—and never setting a disk on a smaller one—transfer the entire pile to another peg. If you don't have one of these in your closet, here is a pocket program of the puzzle for you and your friends.

When the program is run, four "rings" (they will actually look more like short bars) will appear on the left of the screen. There is room on the screen for three piles of rings. (To make the game pocket-sized, the pegs were left out.) To move a ring from one pile to another, press key 1, 2, or 3 to designate which pile (left, center, or right) to take the ring from, and then press 1, 2, or 3 to designate which pile to move the ring to. That's all there is to it.

The program works this way: rings are represented by the numerals 1, 3, 5, and 7. Peg (1), Peg (2), and Peg (3) are variables in which the presence of rings on the three pegs (or piles) are recorded. Thus in line 200, which is part of the initial setup portion of the program, Peg (1) is given the value 1.357 corresponding to the presence of all four rings on the first peg. The leftmost numeral is the one on top.

At the beginning, pegs #2 and #3 are empty. When a ring is moved from one peg to another, the values of the "peg( )" variables change accordingly. For example, if our first move is to place the top ring from peg #1 onto peg #2, then Peg (1) changes from 1.357 to 3.57 and Peg (2) changes from 0 to 1.

These changes are performed in line 450 (where the "size" of the ring being moved is figured out) and in lines 500 and 510 where the values of the "peg( )"s are actually changed. "From" and "too" identify the pegs. They are given values when the keys 1, 2, or 3 are pressed. The three "top( )" variables are strictly for the graphic display; they record the positions of the tops of the piles on the screen. Conveniently, the rings are 1, 3, 5, and 7 characters wide.

```
100 REM  *********************
110 REM  * TOWER OF HANOI  *
120 REM  *********************
130 REM
140 REM
150 REM
160 REM
170 DIM PEG(3),TOP(3)
180 CALL COLOR(7,1,1)
190 CALL COLOR(8,2,2)
200 PEG(1)=1.357
210 PEG(2)=0
220 PEG(3)=0
230 TOP(1)=10
240 TOP(2)=14
250 TOP(3)=14
260 CALL CLEAR
270 CALL HCHAR(10,6,88,1)
280 CALL HCHAR(11,5,88,3)
290 CALL HCHAR(12,4,88,5)
300 CALL HCHAR(13,3,88,7)
310 CALL KEY(3,FROM,STATUS)
320 IF STATUS=0 THEN 310
330 CALL KEY(3,DUMMY,STATUS)
340 IF STATUS=-1 THEN 330
350 FROM=FROM-48
360 CALL SOUND(100,110,3)
370 CALL KEY(3,TOO,STATUS)
380 IF STATUS=0 THEN 370
390 CALL KEY(3,DUMMY,STATUS)
400 IF STATUS=-1 THEN 390
410 TOO=TOO-48
420 CALL SOUND(100,262,2)
430 IF (FROM<1)+(FROM>3)+(TOO>3)+(TOO<
    1)THEN 310
440 IF (PEG(FROM)=0)+((PEG(TOO)<>0)*(P
    EG(FROM)>PEG(TOO)))THEN 310
450 SIZE=INT(PEG(FROM))
460 TOP(TOO)=TOP(TOO)-1
470 CALL HCHAR(TOP(FROM),3+(FROM-1)*9+
    .5*(7-SIZE),87,SIZE)
480 TOP(FROM)=TOP(FROM)+1
490 CALL HCHAR(TOP(TOO),3+(TOO-1)*9+.5
    *(7-SIZE),88,SIZE)
500 PEG(FROM)=10*(PEG(FROM)-SIZE)
510 PEG(TOO)=.1*PEG(TOO)+SIZE
520 GOTO 310
```

"Status" is used as part of the "call key" routine tell the machine when a key has been released so that the program can go ahead. Now read through the program and see if you can follow what is happening.

Stacks

The piles of rings in this program are particularly graphic illustrations of the *stack*, a ubiquitous and very important idea in practically every kind of software. Like the rings in these piles, things stored on *software stacks* (subroutine return addresses, interrupts, whatever . . .) come off the stacks in reverse order to the way they went on. Because the items stored on our pegs are only single numerals, we are able to use a simple "trick"[1] to represent each of our three stacks. We just construct a number for each digit we want to represent. The 99/4A employs numbers accurate to 13 decimal places using a radix-100 representation, so we can push and pop numerals onto and off the left end of these with abandon, multiplying and dividing by 10 without fear of a roundoff error.

[1] If you find an application for this "trick" in a program of your own, you be entitled to call it a "method." (A "method" is a trick used twice).