This Manual contains information on COLECO'S (tm) SMARTBASIC. Any or part of it may be freely copied, edited, and reprinted in any format provided credit is given to the author for the source material. The chapters generally progress from simple to more complex topics. If you intend to break this up for publication in a newsletter, the information should be presented in the order of the table of contents.

Thanks to all those people out there who are dedicated to the support of the ADAM. Several of them have helped me in one way or another discover more about SmartBasic.

Some of you may be aware that I practically disappeared from the ADAM scene in 1994. Although I still have my ADAM, I use it only rarely. Still, there are a few applications like my cheque book balancing program (CP/M) and my income tax calculator (ADAMCALC) that live on my ADAM and get regular use.

Consider this manual a gift to those serious ADAMPHILES who are still using it. I hope it will help you get more enjoyment out of your ADAM.

This manual is an exploration of all the BASIC commands. I will cover them in related groups and I will illustrate ways of getting more out of certain commands. The explanations and hints should help develop better programming habits for the novice as well as the expert.

Guy Cousineau
1059 Hindley Street
OTTAWA Canada
K2B 5L9

TABLE OF CONTENTS

LOOPING

```
FOR x=1 TO 10
FOR x=a TO a+b
FOR x=10 TO 5 step -1
```

Above are various ways of starting a loop. But what is a loop anyway? It is a series of statements which must be repeated several times. Using loops helps reduce the number of program lines; besides it would be tedious to re-write the same segment of code 20 or even thousands of times for large loops. Consider the following program:

```
10 FOR x = 1 TO 10
20 PRINT x
30 NEXT x
```

As you would suspect, the program would print the numbers 1 through 10 and then stop. X is used as a counter by the BASIC program and the NEXT X instruction returns control to line 10 until such time as X is greater than 10. But what if I want to print even numbers from 10 to 20?

```
10 FOR x = 10 to 20 STEP 2
20 PRINT x
30 NEXT x
```

Note the new STEP instruction in line 10. It tells the program to increase the value of X by 2 on each pass. The STEP instruction can be any positive or negative real number: -3.2, +3.66, .001, are all valid values, you can even use a variable! Note that the STEP instruction is optional and a default value of +1 is assumed if no STEP is given on the FOR line.

Say you want to print a multiplication table for values from 0 to 5. Here you need 2 loops, a loop NESTED inside another loop:

```
10 FOR x = 0 TO 5
20 FOR y = 0 TO 5
30 PRINT x*y
40 NEXT y
50 NEXT x
```

Note the large loop which extends from line 10 to line 50, and the smaller NESTED loop from line 20 to line 40. The second loop will run for values of Y from 0 to 5 and then fall through to the X loop which will ask to repeat the Y loop one more time.

Now for the programming tips. It is not necessary to specify the variable name when giving a NEXT command. BASIC keeps track of the CURRENT loop and will execute it automatically. Furthermore, NEXT instructions will actually run FASTER if no variable is supplied. The multiplication table routine could therefore be written as:

```
10 FOR x = 0 TO 5
20 FOR y = 0 TO 5
30 PRINT x*y
40 NEXT
50 NEXT
```

If you have several nested loops, giving umpteen NEXT instructions may get tedious and take up valuable program space. There is another alternative: put your NEXT instructions in one command:

```
10 FOR x = 0 TO 5
20 FOR y = 0 TO 5
30 PRINT x*y
40 NEXT y,x
```

This program will run just as the others, albeit a trifle slower. The instruction on line 40 means DO A NEXT Y UNTIL THERE ARE NO MORE THEN DO A NEXT X. Note that it is very important to specify the variables in the correct order.

Here's one for advanced programmers. What if you want to break out of a loop? Let's go back to the table above. Say I want to print my multiplication table just for results that are smaller than 10....

```
10 FOR x = 0 TO 5
20 FOR y = 0 TO 5
25 IF x*y>9 GOTO 50
30 PRINT x*y
40 NEXT y
50 NEXT x
```

Note the new line 25 which branches the program to the NEXT X instruction. You don't have to worry about leaving the NEXT Y loop open since SMARTBASIC and most other BASICs will automatically terminate the 'y' loop when it encounters the NEXT X instruction which is higher in the shell:

```
/ 10 FOR x = 0 TO 5
outer| 20 FOR y = 0 TO 5¥
loop| 25 IF x*y>9 GOTO 50 |inner
 | 30 PRINT x*y |loop
 | 40 NEXT y/
 ¥ 50 NEXT x
```

Note that when using this technique you must always specify the variable name in NEXT statements. Use this approach only with extreme caution; it is very easy to get all tangled up.

Routine Addresses:

The execution of FOR starts at 8557 (216D hex) and is very complex. It evaluates the FROM TO and STEP variables in floating point and stores all that information on the STACK. It then places the NEXT execution address on the stack and saves the stack address in register IY. If all this sounds complicated, IT IS! it is therefore not advisable to mess with any part of this code. The NEXT execution ranges from 8749 to 8954 (222D to 22FA hex) and is just as complicated.

DECISIONS AND BRANCHING

Decision making is a very important part of BASIC programs and is often the cause of tedious gymnastics. This article will cover 5 BASIC statements dealing with decisions and subroutines.

The IF statement precedes most definitions. It is followed by a mathematical or logical statement which can be simple or very complex:

```
IF x
IF x = 2
IF x^2 >= 2*(y+z)
```

IF x=2 AND y=3

When BASIC encounters an IF statement it evaluates the expression on the left and compares it to the expression on the right. If the condition is TRUE, the rest of the line is executed. If the expression is FALSE then the rest of the line is ignored. Note in the first example that there is no expression on the right; it is evaluated as X NOT EQUAL TO 0.

The IF statement is followed by THEN and another statement. Let's say I want to make sure that x is never bigger than y:

IF x > y then x = y

Note that IF THEN works on the rest of the physical line and not only on the rest of the statement:

IF x > y then x = y : z = 5

In the line above, z will be set to 5 ONLY when x>y. This approach can help simplify programs and reduce the amount of jumping around required.

Sometimes the amount of work to be done cannot be expressed on a single line; in other cases a decision may be required to determine which routine to execute next. In those cases, the IF statement is followed by GOTO:

```
100 IF x = 1 GOTO 150
110 IF x = 2 GOTO 200
120 IF x = 3 GOTO 250
130 GOTO 300
150 y=5
160 GOTO 300
200 y=22
210 GOTO 300
250 y=77
300 more program
```

Note that the 3 IF statements at the beginning which branch to 3 different areas based on values of x. Note also that THEN is not required: IF...GOTO is sufficient and actually executes faster than IF...THEN GOTO. The example above, however shows how programs can become needlessly complicated. Look at the following version of the same decision routines:

```
100 IF x=1 then y=5
110 IF x=2 then y=22
120 IF x=3 then y=77
300 more program
```

We have just taken a 10 line program and changed it to 4 lines which look neater.

Let's consider something a bit more complex like MENU decisions. Assume we have 5 major functions which are selected by entering a number from 1 to 5. Here's where ON...GOTO can come in handy. An ON statement looks up a list of line numbers and decides which one to use based on the DECISION variable:

ON x GOTO 100,250,375,465,555

In the example above, the program will jump to line 100 when x=1, to 250 when x=2, and so on. What happens if x is greater than 5? No jump is made and the program falls through to the next instruction. Considering the MENU options selection stated above, the program line following the ON...GOTO could print one of those NASTY messages like "option 1 to 5, silly". Note also that ON

works on INTEGER values only and that it starts at a value of 1. The ON statement can be followed by a complex mathematical formula if required:

```
ON INT(x/3)+1 GOTO 100,200,300,400,500
```

Note also that ON...GOTO will continue with the next statement (not the next physical line. For experienced programmers, this can be used instead of IF...GOTO to concatenate several lines together even if decisions are required. Consider the following line:

```
ON x=2 GOTO 500 :  y=y+5 :  GOTO 700
```

Here, a jump is made to line 500 whenever x=2 (just like IF...GOTO). However, when x<>2, the rest of the line is executed (unlike IF). Note that when 'x=2' is true, a value of 1 is returned thereby executing the first jump in the list; when the expression is false a 0 is returned and no jump is made.

SUBROUTINES can help reduce program size by executing repetitive functions from a central location. Say you have a MONEY program that prints Dollars and Cents. You spend considerable time developing a technique for right-aligning your figures, making sure there are always 2 digits after the decimal, and preceding the value with a dollar sign. This routine will be several lines long and you will want to access it from various areas in the program. What you do is set up the routine at line 1000:

```
999 REM print x in dollars and cents.
1000 IF x=0 GOTO 1100
1010 IF x<0 GOTO 1200
1020 ....
1399 RETURN
```

The routine ends with RETURN which marks the end of the subroutine. Whenever you want to use the routine, you just place the required value in x and GOSUB:

```
150 x=123.45
160 GOSUB 1000
170 more program
```

Line 150 sets the value to be decoded, and line 160 instructs your program to execute the subroutine at line 1000. When the RETURN instruction is reached, the program RETURNS where it came from and continues, in this case, with line 170. Note that GOSUB need not be on a line by itself: the RETURN instruction will go back to the NEXT statement in the line:

```
150 x=123.45:GOSUB 1000 :  x=y/2 :  GOSUB 1000
```

The line above will do just what you expect: It will first print $123.45, then it will print one half of the value of y.

Note also the REM statement BEFORE the start of my subroutine. It is a good idea to remind yourself what a routine does and how it does it. This can be indispensable later on when changing your program; don't let a failing memory disable your programs. Not also that the REM statement is BEFORE the routine and not the first statement in the routine itself. This technique will help make your programs run faster since the REM statement does not need to be read every time the routine is executed.

In some cases, you may want to abort from a subroutine to go elsewhere. Say I am using the routine above to show how much money you have left in a game. If you ever run out of money, I want the program to exit. Here's where the POP instruction comes into play:

```
999 REM print x in dollars and cents.
1000 IF x<=0 GOTO 1100
1020 ....
1100 POP
1110 PRINT "no money left"
1120 GOTO 100
```

The POP instruction at line 1100 tells BASIC to discard the RETURN address it had stored. Then, program control is returned to line 100 where you might check scores, ask to play again, etc. It is essential to POP correctly in order to maintain all your pointers for the same reasons that using GOTO a subroutine will result in RETURN WITHOUT GOSUB.

Remember IF and ON above? They can also be used quite effectively with GOSUB as well:

```
IF x=3 THEN GOSUB 1000:  y=4 :?"Hello"
ON x GOSUB 1000,2000,3000 :  Print x+y
```

In the first example, the GOSUB instruction will be executed only when x=3; upon RETURN from the subroutine, the rest of the line will be executed. In the second example, the subroutine calls will be made for x=1 x=2 and x=3; for nay value of x, however, the x+y value will always be printed. Note that the RETURN from ON GOSUB comes immediately after the line number list.

Routine Addresses:

IF executes at 7705 (1E19H). It starts by reading the equation and skips the rest of the line if the equation is false.

GOTO executes at 8342 (2096H). It gets the line number, finds the line, sets run mode on, and executes the line (if found). Note that GOTO does not initialize variables as does RUN. It is therefore possible to re-enter a program that has crashed by using GOTO and a strategic line number in the immediate mode. Note that the line number chosen must not be inside a loop or in a subroutine.

ON executes at 8381 (20BDH) and does all sorts of gymnastics to determine the entry in the table to be executed. It then branches to GOSUB or GOTO as required to complete the execution.

GOSUB executes at 8427 (20EBH). It saves the RETURN address on the stack as well as the stack address in IX. It then branches to GOTO to jump to the required routine.

RETURN executes at 8477 (21DDH). It simply reloads the STACK POINTER with the address of the control routine and jumps back to it.

POP executes at 8493 (212DH). It checks if a GOSUB was active and re-adjusts the pointers as required. If there was no GOSUB in progress, an appropriate error message is printed.

As these routines are all very complex, it is not advisable to mess with them. The only interesting patch is to allow formulas in GOTO or GOSUB:

```
10 FOR x=0 to 7
20 READ y
30 POKE 8342+x,y
40 POKE 8437+x,y
50 NEXT :  END
```

```
60 DATA 0,0,0,205,3,39,68,77
```

Line 30 installs the patch for GOTO and line 40 for GOSUB. What the patch does is replace the routine that gets a NUMBER from the command line with a routine that evaluates an EQUATION from the command line.

INPUT COMMANDS

There are various ways of getting values into your programs: keyboard, joystick, peeking around, or using the program itself. This article will cover using the keyboard and program...the others will follow later.

INPUT can be used to let the user give a string (word) or a numerical value to the program. It also has the option of printing a message which will be referred to as a prompt:

```
INPUT x
INPUT y$,z$
INPUT "name:  ";n$
```

Above are all valid versions of INPUT commands. The first requests a numerical value, the second asks for 2 strings, and the third prints a prompt then asks for a string. Note that BASIC will not allow you to respond with a string at the first prompt and will give you a nasty re-enter message if you do. Note also that there is a space after the colon in the prompt in the third example. Remember that SMARTBASIC will not automatically add a space after a string and you should provide adequate spacing yourself. Note also that when entering a string it is not necessary to place the string in quotes unless the string includes a comma (,). A quote character (") can never be part of a string entered at the keyboard.

INPUT is not buffered. This means that you can't supply values ahead of time for other inputs that will follow. Try the following program:

```
10 INPUT x
20 INPUT y
30 PRINT x,y
```

When you run the program, type 1,2 at the first prompt. BASIC will respond with an 'Extra Ignored' and re-prompt (at line 20) for another input. This is a minor error which only means that you supplied more values than you were asked for by the command being executed. Now type in 3 at the second prompt and the program will echo 1 and 3 as the values of x and y. If you want to input both values at the same time you should use INPUT x,y.

GET is handy for MENU-TYPE applications when you want to get only one character without having to also hit the RETURN key:

```
10 PRINT "Continue or Quit (c or q)"
20 GET k$
30 IF k$="q" then end
40 IF k$<>"c" goto 10
50 more program
```

In the example above we expect the user to respond with either "q" or "c". Line 40 goes back and asks the question again if the answer was unacceptable. Note that contrary to INPUT, GET cannot be preceded with a prompt; it is supplied via a regular PRINT statement. GET can handle any key press

including the SMART KEYS, SPECIAL KEYS, and even CONTROL-C without affecting program execution. A program that uses GET's instead of INPUT's allows more flexibility and cannot be CRASHED by incorrect input.

If you consult a table of keyboard codes, you can make use of the values returned by MOVE STORE PRINT CLEAR etc to handle menu options. The following example is not the most effective use of this approach but illustrates the concept:

```
100 GET q$
110 q=ASC(q$)
120 a=0
130 IF q=146 THEN a=1 :REM MOVE/COPY
140 IF q=147 THEN a=2 :REM STORE/GET
150 IF q=148 THEN a=3 :REM INSERT
160 IF q=149 THEN a=4 :REM PRINT
170 IF q=150 THEN a=5 :REM CLEAR
180 IF q=151 THEN a=6 :REM DELETE
190 ON a GOSUB 1100,1200,1300,1400,1500,1600
200 GOTO 100
```

The routine above sets values of a based on the 6 function keys on the right of the keyboard. An ON GOSUB instruction is used to execute the appropriate routines. Note that subtracting 145 from the value of q would have worked but you might have also wished to evaluate the values for UNDO HOME etc. which do not have consecutive values.

When using GET to input numbers only, it is interesting to note that the GET function accepts NUMBER-PUNCTUATION such as " . + - e E ". This can lead to interesting gymnastics on the programmers part...try writing a routine that will use GETs to correctly receive a number such as +3.67E-12 or -365.42.

DATA is a handy way of providing reference information to your program. DATA statements can be composed of numbers or strings:

```
1000 DATA 1,2,3,4,5
1010 DATA Guy Cousineau, "Hi, My name is Guy",Ottawa,Ontario
```

The first example contains 5 numbers; quickly now, how many string elements in the second line? The correct answer is four. The space between Guy and Cousineau will be included in one string containing my full name. The second string is enclosed in quotes since it contains a comma. Ottawa and Ontario form the 3rd and 4th strings respectively.

READ is used to enter DATA statements in programs. Using the example above, my program might start with:

```
10 READ a,b,c,d,e
20 READ me$
30 READ hello$
40 READ city$
50 READ province$
```

Note that READ will start at the first DATA statement in sequence and continue forward until there are no more. You may have encountered the OUT OF DATA message; it simply means that you have tried to READ more DATA than there is in the program. Note also that it is very important to read the data in the correct order and not to try to READ a string data into a numerical variable or you will get another nasty message. Note that DATA statements can be located anywhere in the program and need not

be sequential. SMARTBASIC will handle finding them, even if they are not on a line by themselves.

CLEAR does the same thing as RUN; it resets all variables to zero and resets DATA pointers. This is a trick used by some programmers to prevent you from analyzing a program after it ENDs. It can be handy during a program-restart operation. For example, in a game situation, rather than use tedious assignments like:

```
1000 a=0:b=0:c=0:d=0
1010 q=1:r=2
1020 GOTO 100
```

You can replace line 1000 with CLEAR which resets all variables to zero and then reset only the ones that should not be 0.

RESTORE affects only DATA statements. The command resets the pointer to the current data element back to the beginning of the program. There are enhancements that lets you pick the RESTORE line number (as other BASICS allow) but this routine forces the RESTORE command to always be followed by a valid line number...a better patch will be coming one of these days.

IN lets you branch program control to another routine which handles all keyboard requested input. These are very complicated and required a thorough knowledge of the BASIC operating system. This is not a command to mess around with. BASIC is set up to have all these routines pointing to the regular routine so using an IN command has no effect at all. If you are curious, the syntax is:

```
IN#1...IN#8
```

FP and INT were added to be APPLE compatible. Some BASICS allow you to specify if your program is using FLOATING POINT or INTEGERS for input. Since BASIC has implemented this function differently, all these commands do is change the system prompt from ']' to '>' and vice-versa. Integers will be the subject of another discussion.

Routine Addresses:

INPUT executes at 8957 (22FDH). It starts by looking for a prompt message. If there is none, the character at location 9003 (a question mark) is printed; you can change this to anything you want (happy face, asterisk, etc). When a string is entered, it is checked for CONTROL-C and the program aborts accordingly. You can defeat this feature by POKEing 3 zeroes in 9026,9027, and 9028. The INPUT STRING subroutine ignores leading spaces unless they are in quotes. To defeat this feature, POKE a 255 into location 9236.

GET executes at 9378 (24A2H). It requests one character from the keyboard (waits for it) and returns the appropriate string or number.

DATA executes at 8419 (20E3H). It is a very simple routine that just ignores the rest of the statement (not line). The interesting routine is the one that PARSES a DATA line when it is entered. That one starts at 15184 (3DC6H) and is the culprit of the DATA/REM bump bug. Every time you load in a program that has a DATA or REM statement in it, an extra space is added. This can eventually push your DATA off the end of the line...oops. Fix this routine with the following:

```
POKE 15830,8:POKE 15831,55:POKE 15832,19
```

READ executes at 9499 (251BH). It starts by verifying that there is DATA left. It then executes the appropriate subroutine based on string or numerical input. To accept leading spaces in strings that are not in quotes, poke a 255 in location 9618.

CLEAR executes at 8141 (1FCDH). It starts by CALLing RESTORE and clears all pointers to variables.

The values are still in RAM somewhere but BASIC can;t find them any more. If you want to implement a CLEAR without RESTORE, simply POKE 3 zeroes in 8141, 8142, and 8143.

RESTORE executes at 9482 (250AH). It resets the DATA pointers to the start of the program.

IN executes at 12084 (2F34H). It skips over the '#' in the command and extracts the next digit aborting if greater than 8. It then read in the corresponding vector from a table at 16229 and stores the current IN vector at 16197.

FP executes, in a roundabout way at 20419 (4FC3H). The system prompt for FP resides at address 20420 and can be any ASCII character.

INT also executes in a roundabout way at address 20416 (4FC0H) and it's prompt character is at 20417. Both routines store the prompt into location 1146. So you can change the prompt by POKEing a value to 1146 and restore it with a FP command.

BASIC OUTPUT

Odd as it may seem, there is only one way to send out anything from a BASIC program: the PRINT statement. It is, however, a very versatile command:

```
PRINT "hello"
PRINT a$,b$
PRINT a;" dollars and ";b;" cents"
PRINT x^2+5*y
```

Above are all valid print statements. The first prints a message. The second prints 2 string variables. The third mixes variables and string data. The fourth prints the result of the arithmetic formula. In short, you can mix and match any kind of variable and use the PRINT parameters to format your output.

The comma is used to space-out to the next half of the screen and can be used to neatly format data in 2 columns. Consider the following program:

```
10 FOR x=1 to 10
20 PRINT x,2*x
30 NEXT x
40 END
```

As you have already figured out, it will print a multiplication table. But when you run this program, you will notice something peculiar:

```
12
24
36.....etc.
```

All but the first line are indented by one space. This is a result of the arithmetic used to determine if you are in the first column. You can solve this problem by adding:

```
5 PRINT " ";
```

to indent the first entry as well. There is not enough room in the existing code to effect a proper fix.

The character used to fill between entries is normally a space; you can change this by POKEing the appropriate value in 7884. To change the spacing from 1/2 screen to 1/4 screen, make the following changes:

POKE 7879,7: POKE 7881,8and to restore
POKE 7879,15:POKE 7881,16

The ';' is totally ignored by PRINT and its use, in several cases is only cosmetic. As a matter of fact, if you omit it in SOME statements, the PARSER will add it in for you.

The PRINT parser is located at 15580 (3CDC). It looks for ; : , " . numbers letters and equations by calling other parsing routines. There is not much you can change in there.

The entire PRINT execution routine is found from 7800 to 7899 (1E78-1EDB) with the actual entry point at 7854 (1EAE).

The PR command is used to select device output. Similar to IN, this command accepts up to 8 devices with all but PR#1 being the same. The command sets a flag to channel output to an external device (eg the printer). It is possible to patch in other PR vectors to point to drivers for DOT MATRIX printers but they are only of limited value in BASIC. If you really want one, just ask; it should not be too complicated.

DECIMAL ALIGN

Here's a bonus routine. This is my way of compensating for the lack of a PRINTUSING command in BASIC. Use this subroutine in your programs to decimal align your numbers.

```
10 HOME:? "FORMATTING COLUMNAR DATA"
20 ? "By Guy Cousineau"
30 ?:? "See the REM statements in the  program for usage.  The routine"
31 ? "at 1000 can be used in your    programs."
40 ?:?:?
50 ? "input TEN numbers of different sizes (including negatives)."
60 FOR x=1 TO 10:INPUT n(x):NEXT
70 INPUT "column at which to align       decimal point "; c
100 REM send numbers one at a time in x to print at column c
110 FOR y=1 TO 10: x=n(y)
120 GOSUB 1000:REM print x after formatting
130 NEXT
140 END
998 REM subroutine to print numbers.  With a bit of work,
999 REM can be used to print dollars and cents
1000 IF x=0 THEN HTAB c:GOTO 1030:REM can't use LOG of zero
1005 x0=ABS(x): z2=0:IF x<0 THEN  z2=1:REM x must be positive for LOG function
1010 z1=INT(LOG(x0)/LOG(10)):REM get number of digits to the left of decimal
1019 REM move to decimal column minus number of digits minus the MINUS sign 1020 HTAB c-z1-z2:
1030 ? x
1040 RETURN
```

SCREEN FORMAT COMMANDS

The HOME command is used to clear the text window (even the 4 lines in GR and HGR mode). It also places the cursor at the top of the screen. If you want to home the cursor without clearing the screen, just PRINT CHR$(128).

SPEED controls the delay between each character sent to the screen. It can be useful for special effects but should generally be avoided as it infuriates some experienced programmers, fast readers, or anxious game players. Don't try to guess how fast someone can read, fill up a screen and GET a key press to move to the next screen. You can totally disable SPEED with:
poke 12043,195:poke 12044,15:poke 12045,76
Note that these 3 POKEs must appear on the same line or BASIC will crash.

Before discussing INVERSE video, let's look at the way that BASIC handles the screen. There are 2 tables in VIDEO RAM which control characters. BASIC routinely alternate between these 2 tables regardless of the FLASH INVERSE and NORMAL settings. Normally, these 2 tables contain exactly the same thing ie. the characters to be displayed on the screen. If INVERSE is on, both tables are set to the inverse of the character being printed. When FLASH is on, one page has normal text, and the other page has reverse text. For those of you who may have experimented with direct writes to VRAM to place the characters on the screen, you may have already figured out that you must write the character in 2 places if you don't want to get funny results.

INVERSE and NORMAL complement each other. They affect the way that characters are represented on the screen. INVERSE can be handy for printing TITLES at the top of the screen. Note that INVERSE will print regular characters in inverse video and inverse characters in normal video:

```
10 REM inverse demo
20 PRINT CHR$(65):REM this is an A
30 PRINT CHR$(65+128):rem this is an inverse A
40 INVERSE
50 PRINT CHR$(65):REM this is an A
60 PRINT CHR$(65+128):rem this is an inverse A
70 NORMAL
```

The program illustrated above prints an A and an inverse A. It then calls the INVERSE command and does the same thing again. Note the difference.

FLASH is used for emphasis. Depending on the colour selection, it can be very hard on the eyes. For this reason, it should be used sparingly and for short periods. The following program illustrates a good combination of FLASH and INVERSE for emphasis:

```
10 a$="  HEADLINE  ":REM your message here
20 HOME:  FLASH:  PRINT a$
30 FOR w= 1 TO 1000:  NEXT:  REM wait a bit
40 HOME:  INVERSE:  PRINT a$
50 NORMAL:  PRINT "continue"
```

There is one strange thing you can do with FLASH. Try POKEing powers of 2 in address 17006 (16,32,64) or any other number for that matter.

TEXT is the powerful command. Besides being used to exit graphics modes, it can be used to clear or set certain other screen parameters. The routine jumps around a bit but essentially starts at 18453 (4815H) where it resets the cursor value and the BLANK SPACE value, sets the cursor to FLASH mode, and clears FLASH/INVERSE. It then jumps to 17406 (4296H) where the VPD is initialized. You can patch in your default TEXT attributes as follows:

```
17054492Eborder colour
1706042A4background colour
1711542DBnormal character colour
1712642E6inverse colour
```

17164430Ccharacter to fill the screen with
171754317character to fill alternate screen with
17198432Enumber of lines
17199432Fnumber of columns
172014331home line number
172024332home column number

Every time you give a TEXT command, it is reset to the values in the addresses shown above. Be sure these values are the ones you want to live with. To temporarily reset the margins, POKE the values you want in the following addresses; you can then reset using TEXT.

169934261number of lines
169944262number of columns
169954263home line
169964264home column
16956423Cleft margin
16957423Dright margin
16958423Etop margin
16959423Fbottom margin

There are a few other interesting routines that you may like to experiment with:

17275 (437BH) is the start of the GET CHARACTER sequence which flashes the cursor while waiting for a key press. Address 17291 (438BH) contains the wait value between flashes.

18112 (46C0H) scrolls the screen up one line. You may call this routine directly to push text off the top of the screen.

7884 (1ECCH) contains the ASCII value of the character printed between COMMAS in PRINT statements.

SCREEN POSITION COMMANDS

SMARTBASIC has a variety of commands that control the position of the cursor on the screen. Though some may appear similar, each has its own features which makes it different from the rest:

VTAB places the cursor on a specific line without affecting its horizontal position. You can use a formula to determine a VTAB position; the equation does not need to return an integer: "VTAB x/3" is a perfectly valid statement provided it returns a value between 1 and 24 inclusive.

VPOS tells you where the cursor is. This function seems to have limited value but I can think of one possible application in a game-type environment. VPOS and HPOS can be used together to form the equivalent to the SCRN function in GR mode. Say your TARGET is at 10,10. You can check if the cursor is there with:
IF VPOS(0)=10 and HPOS(0)=10 THEN END
Note that VTAB ranges from 1 to 24 and that VPOS ranges from 0 to 23. This is a silly arrangement which cannot be easily corrected.

HTAB is similar to VTAB but it places the cursor to a specified horizontal position from 1 to 31. In combination with VTAB it can be used to place the cursor anywhere on the screen. It can even be used to write outside the scrolling window. Suppose you have a program that uses the top 2 lines for a

title and those 2 lines have been frozen from scrolling (see previous article on screen commands). A HOME command will send the cursor to line 3. A VTAB 1 command will, however, send the cursor to the title line and allow you to update it. If this update sequence is followed by a HOME command, the cursor will be returned to line 3.

The TAB command will SPACE-PAD between the present cursor position and the new position. This function differs from HTAB in 2 essential areas. TAB does not back up; if the cursor is at position 15 and you issue a TAB(10) command, the cursor will stay at position 15. TAB also erases the characters between the present and target positions. The following program illustrates these features:

```
10 TEXT
20 VTAB 10:?"Hello There you all"
30 VTAB 10:HTAB 5:?" Again";:VTAB 11:?
40 VTAB 15:?"My Name is Guy"
50 VTAB 15:?TAB(16);"What's yours"
```

Line 30 replaces "There" with "Again" without disturbing the rest of the text on the line. Note the VTAB at the end of the statement to remove the cursor from the line...otherwise the "you all" disappears. Try removing the VTAB 11. Alternately, you can VTAB 11 HTAB 1 to reposition the cursor at the beginning of a line. Lines 40 and 50 try unsuccessfully to print 2 messages on the same line...the TAB command on line 50 erases the first 15 characters.

The SPC function will space ahead the specified number of spaces from the current position. Note that it must be followed by a semicolon in order to be effective. SPC differs from TAB in that it always advances and wraps around to the next line if required. It can be handy for right-aligning figures or strings:

```
10 FOR x=1 TO 8
20 READ x$
30 PRINT SPC(30-LEN(x$);x$
40 NEXT x
50 END
60 DATA all,these,words,are,aligned,on,the,right,
```

The POS function tells you where the cursor is on the line. Similar to VPOS, it returns a number from 0 to 31 (instead of 1 to 32). It can be used for a variety of checks and, in conjunction with other positioning commands, control the position of the cursor, end-of-line-wrap, and screen scrolling. One example will suffice:

```
10 DIM x$(10)
20 ?"Input 10 long words"
30 FOR x= 1 TO 10:INPUT x$(x)
40 ?"Now to print them with wrapping"
50 FOR x= 1 TO 10
60 IF POS(0)+len(x$(x))>31 then PRINT
70 PRINT x$(x);" ";
80 NEXT x
```

Line 60 checks that the new word will fit on the current line; if not, the PRINT statement effects a Carriage Return. Note that we only need to check the length of the new word and don't need to include the space which follows it (in line 70). If that extra space moves to a new line because of the screen wrap, POS(0) will tell us we are on a new line on the next pass. If this seems unclear...type out the program, try it out, and change some of the values.

The VTAB execution routine is at 11330 (2C42). It gets an equation from the input line and checks

that it is between 1 and 24. If the value is OK it jumps to the cursor positioning sequence at 26219 (666B).

HTAB executes at 11320 (2C38). It gets an equation and does NOT check the range. It then jumps to the cursor sequence at 26191 (664F). This is the one that prevents tabbing past 31 in the 40 column mode. It calculates the horizontal position based on modulus 32 (the remainder after dividing by 32) but uses the WHOLE number supplied to increment the vertical line accordingly. Some simple patches include POKEing a 63 in address 26198 (6656). This will allow you to HTAB up to column 63 (you need some discipline here). Since this function will also increase the vertical position, you MUST issue a VTAB command after a HTAB in order to correct the count. If you have a 40 column mode, try this:

```
10 VTAB 10:  HTAB 35:  ?"1";
20 HTAB 35:  VTAB 10:  ?"2";
30 VTAB 20:  HTAB 1:?"Can you see a 1 and a 2?"
```

POS executes at 10844 (2A5C). It checks that the 'function syntax' was used, gets the cursor position via 26177 (6641) and returns the value in the floating point accumulator.

VPOS executes at 10857 (2A69). It checks that the 'function syntax' was used, gets the cursor position via 26184 (6648) and returns the value in the floating point accumulator.

MISCELLANEOUS COMMANDS

This section covers a few elementary commands which don't readily fit in any other category. This will be the last series before we start the really big stuff.

When you work with arrays (matrices), you must define the size of your array via a DIM statement. Note that small linear arrays (up to 10) do not need a DIM statement. You can freely use variables x(0) up to x(9). Yes ZERO is a valid array definition and should be kept in mind when defining large arrays. Say you want to create a 4 by 13 matrix for storing a deck of cards:

```
10 DIM c(4,13)
```

works but actually creates an array of 5 by 14 (0 to 4 inclusive by 0 to 13 inclusive. The number of array elements is 70 instead of the 52 required. So what? You are using 35% more memory than you really need. A few large arrays combined with a large program can quickly eat up all your RAM space.

This is a good time to introduce INTEGER variables. Ever see a program with lines like c%=a%+2*e%? The PERCENT sign tells smartbasic that your numbers are signed integers in the range of -(2^15) to 2^15. When these variables are stored in memory, each takes 2 bytes compared to a floating point number which takes 5 bytes. Consider the amount of memory required by the following arrays:

```
ARRAYHEADERRAMTOTALDIFFERENCE
DIM a(4,13)53503550
DIM a(3,12)526026575%
DIM a%(4,13)514014540%
DIM a%(3,12)510410930%
```

Not all numbers can be defined as INTEGER variables. DEF, and FOR require a real variable (floating point) because of the nature of their execution. You can, however change an integer variable into a real one with:

```
a=a%(both a and a% are different variables)
```

You can define several arrays on the same line if they are separated with commas:DIM

a(23),b(2,12),c(550,d(2,3,8).

DEF is used to define a function. Unless your functions are complicated, this is not a recommended approach to programming since it makes program logic hard to follow. Let's take a simple example:

```
10 DEF FN cost(amount)=amount*unitcost
20 unitcost=2.34
30 INPUT "Amount to buy ";amount
40 PRINT "Your cost is $";FN cost(amount)
50 GOTO 30
```

This program starts by defining a function which takes the parameter supplied in brackets (amount) and multiplies it by a fixed variable "UNITCOST". Line 40 could have been replaced with:

```
40 PRINT "Your cost is $";amount*unitcost
```

So why use FN? Say you want to deal out random numbers of a varying range and you want your random numbers to be integer values starting at 1. You might get fed up of typing: INT(rnd(1)*10)+1 and occasionally forget a bracket and get a syntax error:

```
10 DEF FN ran(range)=int(rnd(1)*range)+1
20 INPUT "Range ";r
30 PRINT FN ran(range)
40 GOTO 20
```

REM statements are very useful for beginners and even for advanced programmers who want to distribute copies of their programs. A REM statement can be used to describe what a subroutine does or the purpose of a particular program segment. They can also be used to tag program areas where potential bugs exist to remind you where you need to do extra work. One important thing to remember. DO NOT 'GOTO' a REM statement!!! At some point in time you may remove some extraneous REM statements and crash your program with an UNDEFINED STATEMENT error. If you have a subroutine at 1000, insert your REM statement at line 999 to describe what it does. You will often see programs which start out with several REM statements to describe a program, or issue a copyright notice.

LIST is used obviously to list out part of your program on the screen (or printer if PR#1 has been used). It has a somewhat loose syntax:

```
LISTlist everything
LIST 100,200list from 100 to 200 inclusive
LIST 100-200as above
LIST 100-list from 100 to the end
LIST -200list everything up to 200
```

To pause a long list, you can use CONTROL-S. You can resume the list with any key press even CONTROL-S. It is easier for clumsy typists to tape the CONTROL-S several times in succession to start and stop the list. Note also that LIST can be used within a program....think of a use for it.

DEL is used to delete a line number or a range of line numbers; it uses the same syntax as LIST. Do not confuse it with DELETE which is used to delete a file from tape/disk. Delete can also be used within a program. Consider the following situation:

```
10 PRINT "Please wait"
20 LOMEM:30000
30 PRINT CHR$(4);"BLOAD data"
60 PRINT"Program Ready"
```

This program starts by loading some data from file and then proceeds to execution. What if the program crashes and you have to type RUN to start again? You will have to wait those several extra seconds while the data loads back in again. Why not add a few more lines:

```
40 DEL 30
50 REM line 30 is PRINT CHR$(4);"BLOAD data"
```

That way, if the program crashes, you can re-enter quickly with RUN. If it crashes badly enough that you need to reload the data, you can check out the name of the data file with LIST 50. If you use this technique, be sure that line 30 is really there before you save program changes.

You don't need to use DEL to delete one or 2 lines, just type the line number to delete and follow with RETURN. As a matter of fact, this approach is safer since it deletes only one line at a time.

The DIM execution starts at 6942 (1B1E). It checks for double-defined variables, and makes sure that there is enough free space for both the pointer table and the data. If the DIM array is anything but a string array, that RAM area is blocked off so it can't be overwritten. STRING arrays are not blocked off since we can't predict the length of strings.

The parser for DEF starts at 15125 (3B15). It checks that the word 'FN' follows the DEF statement and that 'FN' is also followed by a space...the syntax is critical here. You can change the 'FN' word to any 2 characters by POKEing them into 15152 (3B30). It then continues with another parser which gets an equation in REAL variables (not integer). DEF executes at 8244 (2034). It goes through some interesting gymnastics to find the '(' and extract the number or variable from the brackets, skips the ')', and aborts if anything appears to be wrong. Then the address of the DEF FN routine is passed to the program controller which takes over as soon as the rest of the line is read in (and ignored.)

LIST executes at 7407 (1CEF). It jumps around checking for the variety of command forms and lists lines one at a time while checking for CONTROL-S or CONTROL-C.

DEL executes at 7555 (1D83). It jumps around doing the same thing as list (in a different way) and proceeds to delete the specified lines issuing error messages if not found.

READING THE JOYSTICKS

The joysticks or paddles can be read via the PDL command. PDL is a versatile command which can return several sets of values depending on the application. The first controller is assigned odd PDL numbers, and the second controller is assigned even PDL numbers. For this purpose, 0 is treated as an even number which complicates things since the equivalent SECOND controller value is one less than the FIRST controller value. This article will concentrate on the PDL functions of CONTROLLER 1.

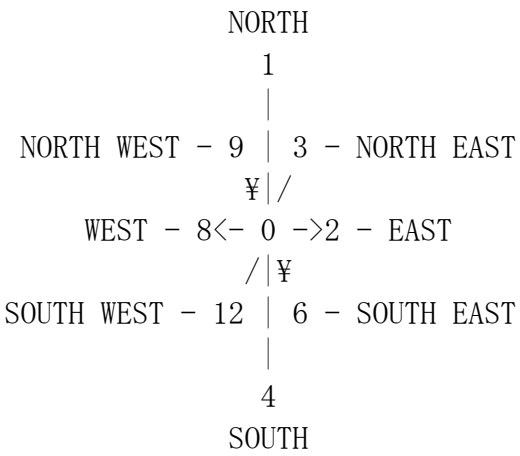
```
PDLUSEVALUES
1vertical counter0-255
3horizontal counter0-255
5directionsee diagram
7left fire button0 or 1
9right fire button0 or 1
11keypad valueASCII 0-9 and '#' '*'
13keypad value0 to 15
15roller controller?
```

PDL(1) updates a counter for the vertical position. If the joystick is pushed up, this value decreases to a limit of 0. If the joystick is pushed down, the value increases towards 255. If the joystick is still or pushed left or right, nothing changes. In a game environment, you must issue a

PDL command every time you want this position to be checked. If you want to reset the game and centre the PDL, you can POKE a 127 into memory location 27100 (27102 for the second joystick).

PDL(3) updates a horizontal counter in a manner similar to PDL(1): LEFT decreases and RIGHT increases. The memory address is 27101 (27103). A combination of PDL(1) and PDL(3) can update the x,y coordinates of the PLAYER. Every time you issue a PDL command, all counters are updated. This may produce undesirable results when you call the function twice to get an x,y coordinate. Furthermore, the 0-255 range for the vertical position is unsuitable since it ranges higher than the vertical height of the screen in HGR mode. For these reasons, PDL(5) is recommended for game applications.

PDL(5) gives a CARDINAL reading based on the 9 positions of the joystick. The rest position returns a 0 and the others as follows:



From the diagram, you can see that NORTH EAST SOUTH WEST are assigned the values 1 2 4 8 and that the intermediate points are the sum of the corresponding values. Let's consider a Game application where we want to update the x,y coordinates of the player based on the joystick position. The first thing to do is get the PDL value in a variable so that the same unique value is used for all calculations:

```
100 p=PDL(5)
```

The next task is to decide whether to increment or decrement the x or y coordinate. We will do this using complex logical equations:

```
110 y=y+(p=4 or p=6 or p=12)
120 y=y-(p=1 or p=3 or p=9)
130 x=x+(p=2 or p=3 or p=6)
140 x=x-(p>7)
```

In line 110 the logical in brackets checks if the PDL is SOUTH SOUTH-EAST or SOUTH-WEST. If either of these conditions is true, the expression returns a 1 which is added to the value of y; if false, a 0 is returned. Note that WEST is easier to decode since all acceptable values are greater than 7.

The next step is to make sure our PLAYER does not fly off the screen by making sure the x,y coordinates do not exceed the playing surface. For our sample program, we will set a box from 50,50 to 200,100. We could use something like IF X>200 THEN X=200 but in a game application, SPEED is often important and we want to take the least amount of computer time possible to evaluate the position. We will therefore use another logical and append it to the lines shown above:

```
110 y=y+(p=4 or p=6 or p=12)*(y<100)
120 y=y-(p=1 or p=3 or p=9)*(y>50)
130 x=x+(p=2 or p=3 or p=6)*(x<200)
140 x=x-(p>7)*(x>50)
```

In line 110, 'y<100' will return a 1 if true and a 0 if false. Thus even if the first logical returns a 1, the value of y will not increase if it is already at 100. Following is a complete demo program which moves a shape to the x,y coordinates chosen:

```
10 REM PDL demo
20 HGR
30 HCOLOR=7
40 SCALE=1:REM set the default size
50 ROT=0:REM make the orientation vertical
60 x=75:y=75:REM set the start position
70 FOR z= 1 to 1000:REM repeat 1000 times then exit
80 XDRAW 1 AT x,y:REM erase the player at last position
100 p=PDL(5):REM read the joystick
110 y=y+(p=4 or p=6 or p=12)*(y<100)
120 y=y-(p=1 or p=3 or p=9)*(y>50)
130 x=x+(p=2 or p=3 or p=6)*(x<200)
140 x=x-(p>7)*(x>50)
150 DRAW 1 AT x,y:REM show the new player position
170 NEXT z
180 PRINT "end of demo"
200 REM PDL demo
```

PDL(7) reads the left fire button and returns a 1 or 0. You can easily check it's value and make a decision with something like:

```
160 IF PDL(7) GOTO 180
```

This additional line in the above program will let the user abort by pressing the left fire button. If it is pressed, the IF PDL(7) returns a true response (1) and branches to line 180.

PDL(9) reads the right fire button. If we want to abort if either button is pressed, we can change line 160 to the following:

```
160 IF PDL(7) OR PDL(9) GOTO 180
```

PDL(11) reads the joystick keypad and returns the ASCII value of the key pressed. The # and * keys will return those corresponding values. In order to get that value into a string expression, you can use the following:

```
a$=CHR$(PDL(11))
```

PDL(13) also reads the keypad but returns numerical values equivalent to the key pressed. If you want to select OPTION 1 or 2 via the keypad you can use something like:

```
100 p=PDL(13)
110 IF p=1 GOTO 1000:REM option 1
120 IF p=2 GOTO 2000:REM option 2
130 GOTO 100:REM wait until valid response
```

Note also that the # symbol returns the value 11, the * returns a 10, and no key pressed returns a 15

(can't use 0 since '0' is 0). It is also possible to simulate the BLUE and PURPLE super controller values which behave as follows:

CONTROLLERPRESSVALUES
purple* and 312
blue# and 313
both# and *14

In order for these values to be read correctly, you must also change the translation table in the EOS with POKE 57861,14; be sure and reset the POKE limit before doing this.

PDL(15) is used to read the ROLLER CONTROLLER. As far as I know, SMARTBASIC does not make use of this function. Special decode routines are required to make full use of the return values. Can someone out there enlighten us?

As you can see, the PDL command is very versatile. It is essential to use it correctly for good program control. The joysticks are not dynamically updated. You must issue a PDL command in order to update the memory values. Each time a PDL command is given, ALL the values are updated in the EOS. The locations illustrated below are updated by BASIC only when a particular command is used. The combinations are complex and will not be discussed here. Experiment with them and draw your own conclusions:

PDL VALUEMEMORY ADDRESS
027100
127102
227101
327103
416788
516783
616799
716784
816780
916785
10not maintained
11not maintained
1216781
1316786

The PDL routine begins execution at 26094 (6918). It begins by updating the EOS controller information and consults a jump table at 27068 (69BC) to determine the correct decode routine. The PDL(0) through PDL(3) functions use another jump table at 27355 (6ADB) to adjust the counters based on direction.

LOW RESOLUTION GRAPHICS

The GR command places you in low resolution graphics mode. This mode is similar to the graphics mode in APPLE BASIC and you can copy some of those programs directly with only minor changes. In the GR mode, the screen in partitioned into a 40 by 40 grid. Each grid can be any colour which gives you reasonable block graphics capabilities.

The COLOR command allows you to set the colour for the plot command. This is similar to choosing a pen colour before drawing something. The default colour is 0 and is always selected when you enter GR mode. Colour values may be set from 0 to 15 which should allow for 16 colours but it doesn't. In their efforts to make SMARTBASIC APPLE compatible, COLECO decided to add a translation table which

converts the COLECO colours to APPLE colours. I have forgotten which colours cannot be accessed, but it is not important. You can defeat the colour translation with the following POKES:

```
POKE 18735,121:POKE 18736,0:POKE 18737,0:REM COLOR
POKE 19256,0 :POKE 19257,0:POKE 19258,0:REM SCRN
```

Now you can use the standard colours where 0 is transparent, 1 is black, and so on.

The PLOT command will paint a block of the chosen COLOR at the x,y coordinates supplied. Say you want to show a single die with the number four. We need to plot four blocks in a square pattern:

```
100 GR
110 COLOR=7:REM choose a colour
120 PLOT 10,10
130 PLOT 10,12
140 PLOT 12,10
150 PLOT 12,12
```

Now we have the four dots, but what about drawing a box around them. I will almost certainly mess it up if I try to plot a whole bunch of points. That's where HLIN and VLIN come in handy. Both commands have a similar syntax and draw horizontal or vertical lines:

```
HLIN from,to AT line
```

Note that 'from' 'to' and 'line' can be integer values or variables. Back to the die program. Let's change the colour of the die outline and continue with:

```
160 COLOR=13:REM change colour for the border
170 HLIN 8,14 AT 8:REM a line 2 wider 2 above the die
180 HLIN 8,14 AT 14:REM same thing below the die
190 VLIN 8,14 AT 8:REM left side
200 VLIN 8,14 AT 14:REM right side
```

The SCRN command reads the colour of the selected grid position. This can be handy in a game situation where you want to know where OBSTACLES, WALLS, TREASURES, BAD GUYS, etc. are. Why bother keeping a separate matrix with all the positions of the players? Use the SCRN function:

```
a=SCRN(5,5)
```

This command assigns the COLOR value of grid position 5,5 to the variable a. This variable can now be checked against the known colours of walls, obstacles, other players, etc. Let's take a simple game in which we will use the joystick manipulation program illustrated in the previous article. We will start the program by drawing 10 OBSTACLES at random on the screen. The player will start in the upper left corner and will be allowed to move freely around the screen. If he should try to move on to an obstacle block the program will abort. Nothing fancy, but it illustrates the basic technique.

```
10 REM PDL and GR demo
20 GR
30 COLOR=12
40 x=0:y=0:REM set the start position
50 FOR z=1 TO 10
60 PLOT INT(RND(1)*30+5),INT(RND(1)*30+5):REM plot one random block
 70 NEXT z
80 COLOR=7:REM set player colour
90 PLOT x,y:REM show where he is
100 x1=x:y1=y:REM remember where player was
```

```
110 p=PDL(5):REM read the joystick
120 y=y+(p=4 or p=6 or p=12)*(y<39)
130 y=y-(p=1 or p=3 or p=9)*(y>0)
140 x=x+(p=2 or p=3 or p=6)*(x<39)
150 x=x-(p>7)*(x>0)
160 IF SCRN(x,y)=12 GOTO 300:REM we hit one
170 COLOR=0:REM set colour to OFF
180 PLOT x1,y1:REM erase player at old position
200 GOTO 80:REM show new position and continue
299 REM hit an obstacle
300 PRINT chr$(7):REM wake player up
310 PRINT "You Lose"
320 END
```

As you can see this is not a fancy program. There is no scoring and no timer. As a matter of fact, the only way to end the program is by running into an obstacle or pressing CONTROL-C. Note that the player is erased just before being redrawn. This will create a short flash so you can see where you are. If the player had been erased at line 95 when we still knew where he was, he would have been OFF longer than ON and would have been difficult to see. This is an important point to consider in game animation: Make sure the bulk of the computer time is used up while the player is ON.

I have seen a few programs that save the contents of a GR screen. This seems like such a waste since it can be simply accomplished with the following:

```
10 DIM m(39,39)
......
799 REM save game and exit
800 FOR x=0 TO 39
810 FOR y=0 TO 39
820 m(x,y)=SCRN(x,y)
830 NEXT y
840 NEXT x
850 REM save the x,y matrix to file or printer
...
899 REM reload the game
900 REM read the data from file
....
950 FOR x=0 TO 39
960 FOR y=0 to 39
970 COLOR=m(x,y)
980 PLOT x,y
990 NEXT y,x:REM remember this one?
```

TECHNICAL NOTES

The screen in GR mode is 256 by 160 pixels. Each of the GR blocks is 6 wide by 4 high which gives 240 by 160. So if you were to paint the whole screen by a series of HLIN or VLIN commands, there would be a BAR on the right side which could not be painted. That is a minor problem as the bar can be considered part of the border; see below for changing the border colour.

The more important point to consider is that the blocks are not square. In order to plot a square block, you need 2 wide (2*6=12) by 3 high (3*4=12). Using this technique, you can still create a 13x13 matrix of square blocks which is reasonable for several GAME type applications.

Remember the colour-bleeding problem? The colour can only be specified for each section of 8

horizontal pixels. So why do a series of 6-pixel blocks not bleed when painted different colours? This is dues to a complicated algorithm that is used to paint part of the block on the foreground and part of the block on the background plane. I have not bothered entering into a detailed analysis of these plotting functions; interested hackers may analyze the PLOT routine to discover how it is done.

ROUTINE ADDRESSES

The GR command executes at 18492(483C). It sets the GR MODE flag, turns INVERSE off and resets the cursor and space values to high-bit characters. It then sets up the GR blocks with a series of character patterns of 6 bytes on and 2 bytes off. The GR text window is set to 4 lines at the bottom of the screen. Following are the POKE values required to change the window size:

number of lines18536(4868)18539(486B)
4320
5419
6518
7617
8716

The default screen colour on initialization is set according to the value at address 18633(48C9) which corresponds to 16*colorvalue. The character colour is set according to the byte at 18711(4917) which corresponds to 16*set colour+clear colour, similar to text colour.

The COLOR command is parsed at 15926(3E36) and 14875(3A1B). The 2 routines look for the symbol "=" and a numerical value respectively. Since COLOR can be specified as a variable, it is not checked for valid input. The routine executes at 11099(2B5B); if the value is 15 or less, the routine continues on to address 18735 where the colour is translated and placed in memory. To disable the COLOR value check, POKE 11107,255. Try plotting blocks with colours above 15 for strange results.

PLOT is parsed in several sections where the line is checked for a number, then checked for a comma, then checked for a number. The CHECK FOR NUMBER parser is at 14875(3A1B). It executes at 11139(2B83) where it first checks that both coordinates are in the range of 0-39. It then jumps to 19102(4A9E) where the type of block to plot is calculated.

HLIN is parsed in 5 segments: number 14875(3A1B), comma 15939(3E43), number, 'AT' 15977(3E69), and number. The routine executes at 11170(2BA2) where it checks that the FROM TO AT values are in the proper range. The rest of the routine is located at 18805(4975); it sets up a loop to call the PLOT routine for each element of the line.

VLIN is parsed in an identical fashion to HLIN. It's execution is at 11219(2BD3) and 18940(49FC).

SCRN is a variable command and the parsing routine resides in an ambiguous part of memory like PEEK, TAB, SIN, etc. It's execution, however is in a fixed position at 11268(2C04). Since this is a MATH function, it gets the first coordinate from the Floating Point Accumulator, checks it for overflow and then gets the second coordinate from the token line. It then calls the translation routine at 19195(4AFB) and returns the COLOR value to the program via the Floating Point Accumulator.

HIGH RESOLUTION GRAPHICS

High Resolution Graphics can be invoked in 2 fashions. The HGR command leaves a few lines for text at the bottom and provides a DRAW window of 256 by 160. The HGR2 command leaves no text window so the full 256 by 192 screen is available for drawing.

HCOLOR, similar to COLOR sets the default pen colour for drawing. As with the GR mode, the colour

value is checked for values between 0 and 15, translated and placed in the table. In HGR, it is highly recommended to disable the check and translation; see HPLOT for applications. The value check is disabled with a POKE of 255 at 11127(2B76). The translation check is disabled by POKE 18747(493B),201.

The HPLOT command is similar to the PLOT command in GR; it turns on one pixel based on the selected colour. Although you can draw shapes in any colour you want, the colour bleeding problem rears it's ugly head here again. One problem which can partially be solved is the bleeding problem when neighbouring pixels are turned off and on. Try these DEMO programs after fixing the HCOLOR check and translation illustrated above:

```
10 REM bleeding demo
20 HGR
30 PRINT"draw a box"
40 HCOLOR=5
50 FOR y=100 TO 150:FOR x=100 TO 150
60 HPLOT x,y:NEXT:NEXT
70 PRINT"Now erase the centre"
80 HCOLOR=0
90 FOR y=110 TO 140:FOR x=110 TO 140
100 HPLOT x,y:NEXT:NEXT
110 print "draw a diagonal in the centre"
120 HCOLOR=10
130 FOR x=110 to 140
140 HPLOT x,x:NEXT
150 PRINT"ooppss"
```

This program proceeds ok to draw a box and erase the centre. When it comes to drawing the diagonal, the other neighbouring pixels are still LIVE and get turned on when the diagonal is drawn. We can borrow the technique used by the XDRAW command to partially cure this problem. an HCOLOR value over 128 (high bit set) will instruct PLOT commands to turn off pixels rather than paint them a different colour. Try this demo:

```
10 REM partial fix of bleeding
20 HGR
30 PRINT"draw a box"
40 HCOLOR=5
50 FOR y=100 TO 150:FOR x=100 TO 150
60 HPLOT x,y:NEXT:NEXT
70 PRINT"Now erase the centre"
80 HCOLOR=128
90 FOR y=110 TO 140:FOR x=110 TO 140
100 HPLOT x,y:NEXT:NEXT
110 print "draw a diagonal in the centre"
120 HCOLOR=10
130 FOR x=110 to 140
140 HPLOT x,x:NEXT
150 PRINT"much better"
```

Now we notice a difference when the centre of the blue box is erased: the left and right borders are the same size. Also, when the yellow diagonal is drawn, we get a line instead of a bunch of blocks. Note that there is still some colour bleeding on the ends of the diagonal but that can also be partially cured by choosing even multiples of 8 for the start and end points. In the program above, replace ALL the 110's and 140's with 120 and 136 to see the difference.

There is no HLIN and VLIN command in the HGR mode, but HPLOT is a versatile utility which can draw

complicated shapes in one command in much the same fashion as a hand-drawing is made without lifting the pen. The syntax is: HPLOT x1,y1 TO x2,y2 TO x3,y3 ...etc. There is no limit (except input line length) to the number of successive points which can be drawn. Drawing squares seems to be a problem: they are not square! Try this demo program which sets up a subroutine to draw a square of the specified size:

```
10 REM square subroutine
20 x=50:y=50:  REM upper left corner
30 z=100:REM size of square
40 HCOLOR=15
50 GOSUB 1000:REM draw a square
60 end
999 REM square drawing routine
1000 HPLOT x,y TO x+z,y TO x+z,y+z TO x,y+z TO x,y
1010 RETURN
```

The square is deformed because of the mechanics of plotting in which the pen advances one pixel in the chosen direction but does not turn on the dot until it moves to the next pixel. The trick is then to adjust the corners by +-1 to square things off. That's why we use a subroutine so we only have to do it right once:

```
10 REM square subroutine
20 x=50:y=50:  REM upper left corner
30 z=100:REM size of square
40 HCOLOR=15
50 GOSUB 1000:REM draw a square
60 end
999 REM real square drawing routine
1000 HPLOT x,y TO x+z+1,y
1010 HPLOT TO x+z,y+z+1
1020 HPLOT TO x-1,y+z
1030 HPLOT TO x,y
1010 RETURN
```

Now this one draws a REAL square. Note also the different syntax used in the subroutine. HPLOT commands can be given with only a TO address. In this case, the HPLOT starts from the LAST PLOTTED POINT or the current position. For program clarity, the second example is better but for execution speed, it is better to chain them all into one command.

Although the HGR mode is suitable to complex graphics, it can be quite tedious to map out the entire screen one pixel at a time. That's why we have shape tables. The format of shape tables is complex and will be the subject of further later in this manual. Suffice it to say for now that a shape table can be installed anywhere in free memory and its address POKED into memory addresses 16766 and 16767.

The SCALE command defines the magnitude of the shape. A scale of 1 will advance one pixel for each plot command in the shape table. The scale can range up to 255 for incredibly large shapes which wrap around the entire screen. When the HGR mode is initialized, the scale is set to 255. It is important then to set the scale to the size you want to use. If you don't, your first draw command will write all over the screen and may take several seconds to complete before you can abort it with CONTROL-C.

The DRAW command prints out a shape definition using the current HCOLOR at the screen coordinates provided: DRAW 1 at 100,100. The draw command can also be given without coordinates in which case it will draw starting at the last pen position. Try this program out using the default shape installed in SMARTBASIC:

```
10 REM repeat DRAW
```

```
20 HGR2
30 HCOLOR=15
40 ROT=0
50 SCALE=2
60 DRAW 1 at 100,100
70 FOR x= 1 to 100
80 DRAW 1
90 NEXT x
99 END
```

The XDRAW command works just like DRAW to erase a shape. Simple animation can be achieved by XDRAW and DRAW commands:

```
10 REM animation using shapes
20 HGR2
30 HCOLOR=15
40 ROT=0
50 SCALE=2
60 FOR x=10 to 200
70 DRAW 1 AT x,100
80 XDRAW 1 AT x,100
90 NEXT x
```

Now your shape darts across the screen. Note that for proper animation, the bulk of the processing time (deciding on move direction, obstacles, and other game conditions) should take place between line 70 and 80 so that the PLAYER is on longer than off.

The ROT command is used to specify the rotation of the shape. ROT values range from 0 to 63 representing different angles in a quadrant. All 64 rotations can only be interpreted if the SCALE value is greater than 7 since the nuances in angular displacement cannot be determined at lower values:

```
10 REM rotation demo
20 HGR2
30 SCALE=16
40 HCOLOR=15
50 FOR z=0 to 63
60 SCALE=z
70 DRAW 1 AT 100,100
80 XDRAW 1 AT 100,100
90 NEXT z
```

ROUTINE ADDRESSES

The HGR command executes at 25484(638C). It fixes up the cursor, space, and flash flags and sets up the text window. The character colour is at 25568(63E0) in the format 16*set color+clear colour. The window parameters are as follows:

```
LINES25573(63E5)25576(63E8)
4320
5419
6518
7617
8716
```

The HGR2 command executes at 25370(631A0). Since there is no text window, the only initialization done is the graphics screen initialization which is identical to HGR:

ADDRESSVALUEDESCRIPTION
25431(6357)colorbyteborder colour
25471(637F)17*colorbytedefault screen colour on init
25479(6387)colorbytecurrent HCOLOR value

The HCOLOR command is parsed at 14875(3A1B) where it calls a routine to get a numerical value or equation. The execution is at 11119(2B6F); it gets the value, checks for legal entry, calls the translation routine at 18728(4928) and stores the value in memory.

HPLOT is parsed at 15102(3AFE). It looks for "TO" and then gets a value, looks for comma and another value, then cycles through again looking for another "TO". The routine execution is at 11487(2CDF). It starts by checking for a "TO" token in which case it gets the FORM coordinates from memory. Each x,y coordinate is extracted and checked for overflow. If there is no TO coordinate, a single point is plotted. Otherwise, the vector is extracted and the LINE drawn at 25686(6456). Upon return, the input line is checked for another "TO". The draw line routine basically extracts the character map from memory and flips pixels on based on the direction calculations. This routine is complex and lengthy; it runs up to 26122(660A).

The SCALE command is parsed at 14875(3A1B); similar to HCOLOR. The execution is at 11473(2CD1) where the value is extracted from the token line and passed to the calculation routine at 26333(66DD) which sets the scale value. If the byte is out of range, the scale is set to 255. You can change this if you want by modifying the byte at 26339(66E3). Also, the default SCALE value is located at 16765(417C). Set it to 1 to avoid messy drawings if you forget to set the scale in your programs.

DRAW and XDRAW are parsed in a similar fashion at 14875(3A1B) and 14976(3A80). The first routine extracts the shape number, and the second looks for "AT" and an x,y coordinate. DRAW executes at 11358(2C5E). It gets the shape number and looks for coordinates. If none are found, the DRAW routine at 26317(66CD) is executed. Otherwise, the coordinates are extracted and checked and the DRAW routine at 26588(67DC) is executed. XDRAW works in a similar fashion at 11405(2C8D) and calls XDRAW routines at 26297(66B9) and 26884(6904) respectively.

The DRAW routine at 26317 extracts the LAST x,y from memory and jumps into the main draw routine. This one finds the data address for the shape table and extracts the vector to the chosen shape. The shape bits are then analyzed and bit-mapped on the screen int he chosen HCOLOR. This routine is also quite complex and runs up to 26883(6903). The XDRAW routines simply set the high bit of the HCOLOR value (erase) and call the corresponding DRAW routine.

ROT is parsed at 14875(3A1B) where a value is extracted. The execution is at 11459(2CC3) which gets the value and passes it to the routine at 26344(66E8). This one divides the value (0-63) by 8 to obtain the quadrant value. It then sets the appropriate pointers so the DRAW routine knows which way to turn the shape.

The default shape table resides in memory at address 26574(67CE). It contains the following bytes:

1 0number of shapes (1)
4 0offset to first shape (4)
54plot-downplot-down
63plot-leftplot-left
36plot-upplot-up
36plot-upplot-up
45plot-rightplot-right

```
45plot-rightplot-right
54plot-downplot-down
54plot-downplot-down
63plot-leftplot-left
0end-of-shape
```

Each byte in the shape data contains 2 plot instructions masked into the first 3 bits and second 3 bits. The last 2 bits could contain NOPLOT instructions but are not required for the default shape. More on shape tables further in this manual.

PROGRAM CONTROL COMMANDS

For a change of pace, we will deal here with elementary commands, yet useful if not essential to programs. Rather than technical notes on routine addresses, we will concentrate on making effective use of the elementary CONTROL commands.

RUN is the way to start a program. It calls the CLEAR routine which initializes variables and arrays and begins program execution at the first line of the program. There are 2 other ways, however, to use the RUN command:

```
RUN line_number
RUN file_name
```

The first will begin program execution at the specified line number. It will still clear variables and arrays, but will not execute DIM statements which have been skipped by the line number specification. e,g.

```
10 DIM a(20,20)
20 PRINT "Hello"
30 FOR x = 1 to 20
40 INPUT a(1,x)
50 NEXT x
```

If you enter RUN 30, you will get an error message since the DIM statement at line 10 will not be executed. While RUN # has advantages in some cases, it must be used with care. See STOP below for a better approach.

RUN file_name is the same as the combined statements LOAD file_name and RUN. It also has a disadvantage in that the program pointers and variables are not saved for program re-entry after a crash. e.g.

```
10 x=1
20 y=2
30 INPUT z
```

Save this program under 'test' file name; type NEW; then type 'RUN test'. At the prompt, press CONTROL-C and follow with 'PRINT x'. You will see that the value of x is 0. Now type RUN and repeat the same procedure. This time, the value of x will be preserved on the interruption.

END marks the end of a program. While it may not be required in most programs, it is a good habit to get into. When your programs grow and you decide to add other features and routines, you may wind up adding lines beyond the actual end of the program. You want to avoid nasty messages, errors, crashes, or other thing that may make your program misbehave. Consider this example:

```
10 GR:COLOR=7
20 INPUT x,y
30 GOSUB 100
40 END
99 REM plot a point at x,y
100 PLOT x,y
110 RETURN
```

When you run this program, everything behaves normally; you enter 2 coordinates and the subroutine plots a point at that location provided it is in the range of 0-39. Now remove line 40 and RUN again. After plotting the first point, the program crashes with a 'Return Without GOSUB in line 110' error.

NEW clears all variables and the program under operation. It is essential to use this command before typing in a new program to be sure that line numbers from your previous program do not become intermixed with the new one. It is not necessary to use NEW before LOAD or RUN since both these routines call the NEW routine. IF you have a protected program that you don't want people to fool around with, put a NEW command instead of END where your program terminates. That will help prevent others from analyzing your program and finding clues to the game or problem. DO NOT use the NEW command in test versions of your program since you will find it difficult to debug and improve your program if it disappears every time it exits.

Here's where a back door comes in handy. A back door is a way for the programmer to get inside his program when required. Consider this EXIT situation to a game:

```
1000 PRINT "Play again? (y/n)
1010 GET q$
1020 IF q$="@" then end : REM back door
1030 IF q$="y" or q$="Y" then RUN
1040 IF q$<>"n" and q$<>"N" GOTO 1010
1050 NEW
```

When the game concludes, the participant is asked to play again. When the programmer answers "@", the program ENDs, leaving everything at his disposal. When others answer YES, the program re-executes. IF they do not answer YES or NO, the routine waits for a correct answer. If NO is the answer, the program destroys itself. This is not a very complex BACK DOOR and could easily be broken by pressing every key except 'y' and 'n' until the program exits, but i won't give away all my secrets.

STOP is a handy way to temporarily suspend your program under test. Suppose you want to check certain values before entering a particular subroutine which is causing you problems:

```
99 REM plot subroutine
100 PRINT x,y: STOP: REM check these values
110 PLOT x,y
120 RETURN
```

Every time you GOSUB 100, the x,y coordinates will be printed out along with BREAK IN 100. If the values are out of range, you can check other values by typing 'PRINT a(2,3)' or any other similar command, including LIST. If everything checks out ok, you can just CONTINUE the program with a CONT command. You can make a STOP conditional with something like:

```
100 IF x>39 OR y>39 THEN STOP
```

STOP can also be used to find out if a program reaches a particular point. Consider the following example:

```
100 IF x<0 GOTO 1000
```

```
110 IF x>100 GOTO 1000
```

If you reach the routine at 1000, you may not know why you got there. In order to check this situation, change the second line to:

```
110 IF x>100 THEN STOP: GOTO 1000
```

Now when x>100, you get a break and when you enter CONT, the program resumes.

TRACE can provide a more detailed check of program execution by reporting all the line numbers being executed. Unfortunately it does not set up a VIEW window and winds up writing line numbers all over the screen: that could be a major project for a BASIC re-write. Furthermore, if your program uses cursor control commands like HOME, VTAB, and HTAB, your line number trace will be hard to follow. The command can be useful if used wisely.

NOTRACE turns the TRACE off. These 2 commands can be used together to isolate the routine under review:

```
100 INPUT x,y
110 GOSUB 1000
...
999 PLOT subroutine
1000 TRACE
1010 IF x>39 GOTO 1050
1020 IF x<0 GOTO 1060
1030 IF y<0 GOTO 1070
1040 IF y>39 GOTO 1080
....
1100 PLOT x,y
1110 NOTRACE
1020 RETURN
```

In this example, the trace will only be active while the subroutine at 1000 is being executed. This presumes that all branches of the subroutine eventually wind up at line 1110 to turn TRACE off.

ERROR TRAPPING

If you want a professional looking program which does not crash upon improper input or behaviour, you can use error trapping. You can use complicated error traps to make it virtually impossible to break into your programs.

WARNING! This is an advanced subject and can lead you into serious difficulties and the loss of valuable programs if used incorrectly. Save all your error trapping until a program is completely debugged. Even so, work on a duplicate copy of your program just in case something goes wrong.

ONERR activates the error trapping sequences it will send the program to an error handling routine via a GOTO statement. It is the programmer's responsibility to write an appropriate error routine. ONERR will trap ALL errors (including ^C), except the "Extra Ignored" which is not an error but a warning.

```
10ONERR GOTO 60
20INPUT x,y
30PRINT x/y
```

```
40IF x=0 THEN END :  REM allow the program to terminate
50GOTO 20
60PRINT "division by zero error"
70GOTO 20
```

Try running this program with normal values; everything behaves as expected. Now enter 1,0 and your error handling routine takes over, prints the message, and prompts for input again. Try entering strings (words) instead of numbers, same reaction. Guess our error handler is not too smart. Try entering 0,0 to exit and find that the error handler takes over since it tries to do the division before checking for the EXIT cue. So type 0,1 to end. Now, delete line 40 and run the program again....CONGRATULATIONS, your first unbreakable program!

RESUME is used in an error handling routine to return to the line number that caused the error. Note that it will try to re-execute the statement that caused the error. To prove this point, re-write the program above and substitute 70 RESUME. When you generate an error with input like 1,0 the program locks itself in a loop and keeps reprinting the divide by 0 error and insisting on re-executing line 30.....CONGRATULATIONS, another program that gets out of control and can't be broken! But is it really? Type ^C several times. If you are fast enough, you might be able to catch the operating system off guard and break with a strange message like Break in 17042. But there is no such line number. At this point, you should reboot BASIC. Although any other action may result in a FATAL SYSTEM ERROR, strange things have been known to happen.

CLRERR is used to turn off the error trapping and is an essential companion to ONERR. IF the routine illustrated above is part of a bigger program, you don't want ALL errors to branch to a DIVIDE BY 0 message. Consider the following:

```
10ONERR GOTO 60
20INPUT x,y
30PRINT x/y
50GOTO 80
60PRINT "division by zero error"
70GOTO 20
80CLRERR
90INPUT x,y
100PRINT x/y
110END
```

Run this program and enter 0,0 the first time; the error handler takes over. Now enter a valid value like 4,2 and get 2 for an answer. As we are now at line 90, try entering 0,0 and get the usual BASIC message since the error trapping has been turned off at line 80.

ERRNUM gives error handlers the ability to be a bit more intelligent by interpreting the error. Information on error codes is sketchy but you can augment the table below by running a program WITHOUT an ONERR statement. Whenever an error occurs, type PRINT ERRNUM(0) to find the corresponding error code. If it does not appear in the table, add it in for future reference.

ERRORMEANING

2range error; parameter to large to handle by parser
5end of data in a file read (see 42)
7file not found
8bad read or write to disk/DDP (ambiguous error)
9directory or disk/DDP full
10file is locked in write or delete operation
11bad file name or other syntax in ^D operations
12too many characters following a ^D or too many files open

13file type mismatch; trying to run a binary file
16illegal function in a READ or INPUT statement
22RETURN encountered with no GOSUB pending
42out of data in a READ statement (see 5)
53illegal quantity in STRING operations, or PEEK POKE SPC TAB,etc.
69floating point or integer number to big to handle.
77out of memory (too many nested loops, program too big, etc)
90undefined statement for GOTO or GOSUB
107bad subscript; using values outside the limits of DIM
120same array specified twice in DIM statement
133division by zero
163type mismatch
176string longer than 255 characters
191formula too complex
224using FN (function) with no DEF FN (function definition)
254bad response to INPUT like STRING when number was expected
255a STOP statement was encountered or ^C was pressed

Now back to our first program to make the error handler smarter

```
100NERR GOTO 60
20INPUT x,y
30PRINT x/y
50GOTO 20
60err=ERRNUM(0)
70IF err=133 then print "divide by zero error":GOTO 20
80IF err=255 then print "Program Aborted":END
90Print "Unknown error #";err
100END
```

Now we have an error handler that can make decisions. If the error is divide by zero, tell the user and try again. If ^C was pressed, end the program. If any other error occurs, show the error code and end the program.

Although there are several ways of handling errors, my recommended approach is to make one routine for each critical part of your programs. That way you always know where you are. Although DATA FILE ACCESS will be discussed at a later date, consider this sample program which handles errors in stages.

```
10INPUT "File Name ";f$
200NERR GOTO 50
30PRINT CHR$(4);"Open ";f$
40CLRERR:GOTO 100
49REM handle bad file name and disk I/O error here
50...
1000NERR GOTO 150
110FOR x=1 to 10
120INPUT x:REM from file
130NEXT x
140CLRERR:GOTO 200
149REM handle end of data, syntax, and disk I/O error here
150...
```

Each section of the program has its own error trap and can handle them more efficiently via ERRNUM(0). Note the CLRERR statement at the end of each critical routine.

NOBREAK has been the subject of a few discussions and it seems very few people understand it. Here's

by best interpretation. SMARTBASIC continually scans the keyboard for a ^C and aborts your programs at your request. Using the NOBREAK command defeats this feature, but not entirely. When a program writes to the screen, a ^C will always work. Try the following program:

```
10PRINT "Press control-C"
20NOBREAK
30FOR x= 1 TO 5000:NEXT
40PRINT "It did not work, did it?"
50PRINT "Press control-C"
60GOTO 50
```

During the first loop, ^C is disabled since there is no screen output. When line 50 get executed in a loop, ^C will abort.

So what's the advantage? Since the keyboard is not scanned, you can increase a program's speed (only marginally) by using NOBREAK in your CPU intensive tasks. The best advantages come into play when you use HGR2 mode. Since there is never any screen output, NOBREAK will make a program harder to abort. The best advantage, however, will be TYPE AHEAD!. Consider this program:

```
10HGR2
20NOBREAK
30GET c$
40HCOLOR=VAL(c$)
50FOR x=0 to 255
60HPLOT x,100
70NEXT
80do=do+1
90IF do=10 THEN TEXT:END
100GOTO 30
```

Run this program and press 1 2 3 4 5 6 7 8 in rapid succession and WAIT. You will see the line changing colours as each colour value is interpreted. After 10 iterations, the program ends. Note that without line 90, the program would lock up...another UNBREAKABLE program

BREAK is the opposite of NOBREAK; it reactivates the ^C checking in your program.

ROUTINE ADDRESSES

ONERR executes at 8114 (1FB2). It verifies that a program is running since it is not a valid COMMAND in direct mode. It then extracts the line number from the command line and saves it at address 16126 (3EFE) after setting the ONERR flag.

CLRERR executes at 8313 (2079). It checks for run mode and that ONERR is active. It resets the ONERR flag and reloads the program counter with the line number where the error occurred.

Execution routines for other commands use the area starting at 7914 (1EEA) to process errors. The error code is stored at 16128 (3F00). If ONERR is detected, a jump is made to 8082 (1F92).

ERRNUM executes at 10184 (2A3E). It gets the error code from address 16128 and extracts the corresponding translation from a table at 1463 (05B7). Oddly enough, the first several error numbers are the offset from the ASCII messages table associated with these errors. This table starts at 1152 (0480).

BREAK executes at 6346 (18CA), and NOBREAK at 6351 (18CF). They merely toggle the flag which is used by the keyboard check routine in the command parser. This routine is found at 6190 (182E). Setting addresses 6193, 6194, 6196, 6197, 6198 (decimal) to 0 will totally disable BREAK while increasing program throughput just a bit more than issuing a NOBREAK command.

PLAYING WITH RAM

RAM stands for Random Access Memory. All of the 64K available under SMARTBASIC is RAM area. This means that different values may be placed in any memory location, and modified, as required, by the controlling program. ROM (Read Only Memory) on the other hand, can only be read. One example of ROM are the GAME cartridges. The information on these cartridges is constant. The program contained in them is copied to RAM prior to program execution.

Before looking at commands that affect RAM, let's make a quick memory map of the standard RAM used by SMARTBASIC:

ADDRESSDESCRIPTION

00000-27407 (0000-6B0F)SMARTBASIC program
27407-?? (6B0F-??)Variables stored up from here
??-54272 (??-D400)User program stored down from here
54272-57344 (D400-E000)3 1K buffers for directory and file access 57344-65535 (E000-FFFF)Operating System

The space between the question marks (26865 bytes) is the RAM area available for user programs. As you write a new program, the top of available memory is adjusted down from 54272. When you RUN your programs, the space from 27407 gets filled upward for each variable and string that you define in your program. If the 2 ever meet, you get that nasty OUT OF MEMORY error.

FRE is used to find out how much memory is available. Since FRE is a variable command (similar to the trig functions), it requires a parameter even though it is not used. Thus FRE(0) is the same as FRE(22). When you first BOOT BASIC, you can ask for the amount of free space with:

```
PRINT FRE(0)
25954
```

But 25954 is less than the 26865 described above. That is because some variables are already defined (the variable commands) and take up a bit more than 900 bytes of RAM. The FRE command can be used within programs to check on the available space. When FRE calculates the memory space, it begins by doing some house cleaning in the string space by crunching up strings which are no longer required. Then it reports the available memory. To force this house cleaning operation, add a line like:

```
f=FRE(0)
```

in a strategic location in your programs. This may help prevent the loss of valuable strings when sorting large string arrays.

PEEK is used to report the value in a particular memory location. This can be handy to check on the status of an operation, or to read data. Let's consider one application. I have a program which performs several LONG iterations (like GAME OF LIFE simulations). I want to give the user a chance to abort without having to press CONTROL-C. Since we know that the last key press register is at 64885, we can PEEK that address to detect the abort request:

```
1000 GOSUB 2000:  REM do one generation
```

```
1010 p=PEEK(64885)
1020 if p=ASC("q") THEN END:  REM quit request detected
1030 if p=ASC("r") GOTO 100:  REM restart request detected
1040 GOTO 1000:  REM continue if no option detected
```

Since PEEK returns a decimal value, we must compare (in 1020 and 1030) with a decimal value by comparing to the ASCII value of the option letters. Note that we could have compared with 113 and 114 but ASC ("q") makes it clear that we are checking for the letter q. When execution speed is not critical, this is a highly recommended programming technique.

POKE is the opposite of PEEK; it places a value in a RAM address. POKE 27407,25 for example would place the value 25 in location 27407. This can later be verified with PRINT PEEK(27407). POKEing below 27407 should be done with extreme caution as it will change the SMARTBASIC program. POKEing above 54372 should also be done with great care since this will change the operating system. SMARTBASIC protects against this by checking the POKE limit at 16145-16146 when the POKE command is executed. Novices should not change the POKE limit until they are ready to experiment with the EOS. To disable the POKE limit: POKE 16145,255:POKE 16146,255. Alternately, you can POKE 10120,2:POKE 10121,201 to defeat the check and make POKE run just a bit faster.

LOMEM can be used to reserve RAM area for user routines or DATA. LOMEM commands should be placed at the beginning of your programs since LOMEM also clears variable arrays before adjusting the memory pointers:

```
LOMEM:28000
```

will set aside 593 bytes from 27407 to 27999 in which you can PEEK and POKE to your heart's content. You don't have to worry about overwriting your program, variables, or SMARTBASIC itself. Note that the LOAD, CLEAR, and NEW commands do not affect the LOMEM setting. Programs that set LOMEM abnormally high may cause you OUT OF MEMORY problems later on when loading other programs. Should this ever happen, type NEW and follow with PRINT FRE(0). If you don't have enough memory to load in the new program, reset LOMEM:27407 and LOAD again. As an added precaution, have programs which reset LOMEM return it to its normal setting as part of the exit routine.

HIMEM is similar to LOMEM but it protects RAM above the specified address. HIMEM is complicated to use since it requires knowledge of the size of your program. HIMEM must be located below the end of your program by an offset equal to the number of bytes you wish to reserve. As HIMEM has no apparent advantages over LOMEM, its use is discouraged. Contrary to LOMEM, HIMEM is reset every time a program exits.

The HIMEM execution will check that the HIMEM address is not overwriting the program or data segment of RAM and abort with OUT OF MEMORY. When it is successful, it saves the address at 16109 (3EED) which is the TOP address for numeric variables. It will not report how much RAM has been protected by this change.

CALL is used just like in MACHINE LANGUAGE routines to execute a routine at a particular address. This is an advanced command which should be left to experienced programmers. The CALL routine saves all the program pointers, executes the requested routine, restores the program pointers and clears the accumulator prior to resuming the BASIC program. It is the programmer's responsibility to preserve the stack and/or set up a local stack.

USR is similar to CALL except that it always branches to the USR routine at memory address 16130-16131 (3F02). At BOOT, this address points to a RETURN and no harm can be done by a USR command. USR is similar to CALL in that program pointers are preserved and the accumulator cleared on exit. The advantage of the USR function is its ability of passing a parameter to the user routine. When the USR routine gets control, the DE register points to the function number with the high bit set. Since USR

is a variable command, it requires a variable. Thus the correct syntax is:

a=USR(n)

where 'a' is any legal numerical variable and 'n' is a number from 0 to 255. Since the high bit is set, values from 128 to 255 are the same as 0 to 127. Following is a sample of preamble code to a series of user functions:

```
LDA,(DE);get function
AND7FH;strip high bit
CP3;max function +1
RETNC;function out of range
LDHL,FTABLE;jump table for routines
LDB,0
LDC,A;function number to BC
ADDHL,BC
ADDHL,BC;point to function vector
LDA,(HL)
INCHL
LDH,(HL)
LDL,A;get execution vector in HL
JP(HL);execute it
FTABLE:
DWFN_0
DWFN_1
DWFN_2
```

The & routine is similar to the USR function except that it does not preserve the DE register which is the pointer to current position in the command line. The routine is useful for those applications which will parse a series of instructions from the command line. It is the programmer's responsibility to return the DE register pointing to next instruction and to clear error conditions in the accumulator. & gets its execution address from memory location 16132-16133 (3F04). At BOOT, this points to REM which effectively ignores the rest of the line.

You may often see programmers using & as a REM statement. This practice is not recommended for programs which will be distributed since other people may have installed & routines.

In order to use the & routine, it is necessary to know something about the register use in SMARTBASIC. Although this will be covered later, this is the essential information:

Register DE contains the pointer to the current line being executed
Register C in the alternate set has the number of characters remaining

Following is a sample start up routine for an & function:

```
EXX;use alternate set
LDA,C;get length of line
LDC,0;set line length to empty
EXX;normal set
LDH,0
LDL,A;length to hl
ADDHL,DE;point de to end of line
LD(SAVEDE),HL;save exit value for de
INCDE;skip header
```

```
INCDE;get length byte
LDA,(DE)
INCDE;skip to first character
LDB,A;save character count
;
;go on from here to decode the INSTRUCTIONS in (DE)
;using B as a down counter
;
;all routines must exit to this routine to properly restore register DE ;
EXIT:
LDDE,(SAVEDE)
XORA;make a zero
SCF;set the NO-ERROR condition
RET
```

WAIT is another advanced command which is used to wait for a particular value in a port. Its use requires detailed knowledge of prot operations and status codes returned by peripherals. The syntax is:

WAIT a,b,c

where a is the PORT number, B is the XOR value, and C is the AND value. The WAIT command will get the value from the port, XOR it with B, AND it with C and continue this operation until a non-zero result is obtained. Using WAIT without the proper parameters will effectively lock up your system.

ROUTINE ADDRESSES

FREE executes at 10192 (27D0). It recovers unused string space, and subtracts the top of variables from the bottom of strings to determine the free RAM space.

PEEK executes at 10091 (276B). It gets the memory address and loads its value into the floating point accumulator.

POKE executes at 10104 (2778). It gets the memory address and checks it against the POKE limit. If the address is within range, it places the requested value in memory.

LOMEM executes at 10870 (2A76). It gets the protected address and moves the variable tables accordingly. If insufficient memory, it aborts with a message.

HIMEM executes at 11010 (2B02). It also gets the protected address and shifts tables accordingly.

CALL executes at 10042 (273A). It gets the address, saves the registers, calls the routine, restores registers and return to the program.

USR executes at 10073 (2759). It saves the registers and extracts the routine address from memory. It then jumps to the CALL routine to finish off.

& executes at 10164 (27B4). It saves some registers and extracts the routine address from memory for execution.

WAIT executes at 10126 (278E). It extracts the PORT XOR AND, and cycles through until a non-zero result is obtained.

USR Sample

Following is a simple USR routine with 2 functions. USR(0) performs a TEXT command and prints a

message. USR(1) makes a binary dump. Both routines use some of SMARTBASIC's printing utilities which are described at the end of this chapter. The listing is in assembler format and includes comments after the ";". Assemblers ignore these statements like BASIC ignores REM:

```
;USR DEMO
;POKE 16130,72 and 16131,133 prior to using
;Reset LOMEM to 30000
;and POKE routine in starting at 29000
;
PRINTAEQU 2EDAH  ;print character in A
PRINTSEQU 12110  ;length encoded message IN HL
TEXTEQU 11065  ;text mode routine
;
ORG29000;start assembly here
LD A,(DE) ;get function number
AND7FH  ;strip high bit
JR Z,FN_0 ;was function 0
DECA  ;check if function 1
JRZ,FN_1 ;do it
ADD'0'+1;make function number ASCII
LD(FNNUM),A ;add it to error message
LDHL,FNMSG ;point to error message
CALL PRINTS;print error
RET   ;exit
;
;function 0
;
FN_0:
CALL TEXT ;restore screen
LD HL,THEEND ;bye message
CALL PRINTS ;print it
RET   ;exit
;
;function 1
;
FN_1:
PUSH DE  ;save value in DE
LD HL,DEMSG
CALL PRINTS
POP AF  ;get D into A (trick)
PUSH AF  ;save it again
CALL SHOWA  ;dump D
POP DE  ;get DE back
LD A,E
CALL SHOWA  ;dump E
LD A,13 ;a carriage return
CALL PRINTA ;print it
RET   ;get out
SHOWA:
LD B,8  ;must do 8 bits
SHOWA1:
RLCA  ;get top bit in carry
PUSH AF  ;save A
LD A,'0'  ;start with a zero
ADCA,0  ;make 0 or 1 based on carry
PUSH BC  ;save counter
CALL PRINTA ;print one bit
POP BC
POP AF
DJNZ SHOWA1 ;continue for 8 loops
RET   ;done 8-dump
```

```
;
;messages
;
FNMSG:  DB 23,'function #' ;first byte is length
FNNUM:  DB '0'  ;filled in by error handler
 DB ' undefined',13 ;end of message
THEEND:  DB 30,'Function 0 restores text mode',13
DEMSG:  DB 29,'The DE register in binary is',13
;
 END
```

If you want to POKE this program in, following are the HEX values; you'll have to convert them to decimal yourself.

```
1A E6 7F 28 0F 3D 28 16 C6 31 32 99 71 21 8E 71 CD 4E 2F
C9 CD 39 2B 21 A5 71 CD 4E 2F C9 D5 21 C4 71 CD 4E 2F F1
F5 CD 7D 71 D1 7B CD 7D 71 3E 0D CD DA 2E C9 06 08 07 F5
3E 30 CE 00 C5 CD DA 2E C1 F1 10 F2 C9 17 66 75 6E 63 74
69 6F 6E 20 23 30 20 75 6E 64 65 66 69 6E 65 64 0D 1E 46
75 6E 63 74 69 6F 6E 20 30 20 72 65 73 74 6F 72 65 73 20
74 65 78 74 20 6D 6F 64 65 0D 1D 54 68 65 20 44 45 20 72
65 67 69 73 74 65 72 20 69 6E 20 62 69 6E 61 72 79 20 69
73 0D
```

& Routine Sample

Following is an extract from an & routine I have written as a demo. It is part of a bigger routine which does HEX/DEC/HEX conversions. The listing below only converts HEX to DECIMAL but still requires an H prior to the number to be decoded. After installing this routine you could
type: & hFC2D
and get 64557

```
;
;& routine demo
;must POKE 16132,72 and 16133,113
;set LOMEM to 30000
;and POKE data in at 29000
;
PRINTA EQU 2EDAH  ;print character in A
PRINTS EQU 12110  ;print length encoded message
;
ORG29000
EXX   ;alternate set
LD A,C
LD C,0  ;clear line
EXX
LD H,0
LD L,A  ;length of line to HL
ADDHL,DE  ;point to end of line
LD (SAVEDE),HL ;save DE exit value
INC DE  ;skip header
INC DE
LD A,(DE) ;get length byte
INC DE  ;point to first character
LD B,A  ;copy length of line to B
;
SKIPSP:
```

```
LD A,(DE) ;get a character
CP ' '  ;is it space
JR NZ,DONESKP ;ready to proceed if not
INC DE  ;skip the space
DJNZ SKIPSP ;skip more spaces
JP EXIT ;abort if all spaces
;
DONESKP:
AND5FH  ;make upper case
CP 'H'
JP NZ,EXIT ;used to be jump to decimal
;
;HEX input routine
;aborts if non-number input
;but never reads more than 4 bytes
;
HEXIN:
INC DE  ;skip the H prefix
DECB
JP Z,EXIT ;no characters left
LD HL,0 ;set default output value
LD A,5
CP B
JR C,HEXIN1 ;we have less than 4 characters
LD B,4  ;reset to max
HEXIN1:
LD A,(DE)
OR A
JR Z,HEXDONE ;end if null
CP ' '
JR Z,HEXDONE ;or space
SUB '0'
JP C,ERROR
CP 10
JR C,NUMOK ;we have a 0-9 digit
AND5FH  ;make uppercase
SUB 7  ;make A-F if 10-15
JP C,ERROR ;oops
CP 16
JP NC,ERROR ;oops again
;slide bits left 4 times in HL
NUMOK:
SLA L
RL H
SLA L
RL H
SLA L
RL H
SLA L
RL H
OR L  ;add incoming digit
LD L,A  ;put back in L
INCDE  ;move buffer up one
DJNZ HEXIN1 ;read another digit
;
HEXDONE:
```

```
CALL DECIMAL ;PRINT HL in decimal
JR EXIT
ERROR:
LD HL,SYNTAX ;error message
CALL PRINTS
EXIT:
LD A,13
CALL PRINTA ;add a new line
LD DE,(SAVEDE) ;restore BASIC's pointer
XORA  ;clear error flags
SCF   ;we're done
RET
;
DECIMAL:
PUSH HL
PUSH DE
PUSH BC
LD B,0  ;leading zeros flag
LD DE,10000
CALL SUBDIV
LD DE,1000
CALL SUBDIV
LD DE,100
CALL SUBDIV
LD A,L
JR SUBDV4
SUBDIV:
XORA  ;set out digit to zero
SUBDV1:
SBC HL,DE
INC A  ;add 1 to digit
JR NC,SUBDV1
ADDHL,DE  ;undo subtraction
DECA  ;adjust count
JR Z,SUBDV2 ;digit is 0
LD B,A  ;clear zeroes flag
JR SUBDV3
SUBDV2:
CP B  ;is zero flag on?
JR NZ,SUBDV3
LD A,' '-'0' ;make A a space
SUBDV3:
PUSH HL
PUSH DE
PUSH BC
SUBDV4:
ADDA,'0'  ;make ASCII
CALL PRINTA
POP BC
POP DE
POP HL
RET   ;done one digit
;
SAVEDE:  DW 0  ;space to store DE
SYNTAX:  DB 11,'Bad Syntax',13
;
```

```
 END
```

To POKE this routine in, following are the HEX values:

```
D9 79 0E 00 D9 26 00 6F 19 22 FA 71 13 13 1A 13 47 1A FE
20 20 06 13 10 F8 C3 B6 71 E6 5F FE 48 C2 B6 71 13 05 CA
B6 71 21 00 00 3E 05 B8 38 02 06 04 B7 28 2E FE 20 28 2A
D6 30 DA B0 71 FE 0A 38 0C E6 5F D6 07 DA B0 71 FE 10 D2
B0 71 CB 25 CB 14 CB 25 CB 14 CB 25 CB 14 CB 25 CB 14 B5
6F 13 10 CF CD C2 71 18 06 21 FC 71 CD 4E 2F 3E 0D CD DA
2E ED 5B FA 71 AF 37 C9 E5 D5 C5 06 00 11 10 27 CD DC 71
11 E8 03 CD DC 71 11 64 00 CD DC 71 7D 18 15 AF ED 52 3C
30 FB 19 3D 28 03 47 18 05 B8 20 02 3E F0 E5 D5 C5 C6 30
CD DA 2E C1 D1 E1 C9 00 00 0B 42 61 64 20 53 79 6E 74 61
78 0D
```

STRING FUNCTIONS

This article will deal with string manipulation. We will cover the basic operations and show a few tricks to make your string operations more effective.

As string operations are all VARIABLE COMMANDS, they all require a parameter (string) which is included within round brackets. Note that when a literal string is included, it must also be in quotes. This will become clearer in the examples that follow.

LEN will give you the length of the string. This is sometimes a very useful value which can be used as a loop counter:

```
10 INPUT "Name ";n$
20 FOR x=1 TO LEN(n$)
30 REM perform some manipulation
40 NEXT x
```

ASC will convert a string to an ASCII value. Expanding on the sample program above, we can do the following:

```
10 INPUT "Name ";n$
20 FOR x=1 TO LEN(n$)
30 PRINT ASC(MID$(n$,x,1))
40 NEXT x
```

This program will print the ASCII values of the supplied name. Not very useful, but it demonstrates the syntax of the function. Note that we have nested one string operation as a parameter for the other. This is quite acceptable as long as the correct number of brackets are opened and closed. When you get a syntax error typing one of these types of commands, check the brackets and quotes first, they are most likely the cause.

Let's look at another example which might make better use of the ASC function. A program uses GET statements to accept user input. Sometimes this is a LETTER which can be checked against an action table, and sometimes it is an arrow key or other control key:

```
999 REM get a key and process command
1000 GET Q$:  q=ASC(q$):  REM convert input to ASCII as well
1010 IF q>159 and q<164 GOTO 2000:  REM process arrow keys
1020 IF q> 31 and q<127 GOTO 1100:  REM process standard characters
```

Note throughout these manipulations, the 'q' variable may be manipulated without losing the actual key press which remains in q$. Note also that ASC works on the first character of any string. Thus to check string input for 'y', you can use the following which will be true if y yes yuppi or anything else starting with y was typed:

```
100 INPUT q$
110 IF ASC(q$)=ASC("Y") or ASC(q$)=ASC("y") GOTO 200
110 REM process NO answer
```

```
200 REM process YES answer
```

CHR$ is the opposite of ASC. It converts a number to its equivalent string character. Going back to our sample program at the start; let's use ASC and CHR$ to convert input to upper case:

```
10 INPUT "Name ";n$: o$="": REM set out string to empty
20 FOR x=1 TO LEN(n$)
30 a=ASC(MID$(n$,x,1)): get one character value
40 if a>96 and a<123 then a=a-32: make UPPER if a-z
50 o$=o$+CHR$(a): REM add new character to o$
60 NEXT x
70 PRINT o$
```

Note here the addition of strings. When 2 strings are added together, the second is appended to the end of the first. You can see this in the following example:

```
10 a$="add"
20 b$="ition"
30 PRINT a$;b$: REM show what they look like
40 c$=a$+b$
50 PRINT c$: REM same result
```

VAL is similar to ASC except that it converts an entire number (not just a digit) to a numerical value:

```
10 INPUT"Give me a number "n$
20 n=VAL(n$)
30 PRINT n
40 GOTO 10
```

You can type in positive or negative numbers, even numbers with exponents; the VAL function handles them. Now try typing "32 dollars" and "$32" as a reply. The first come out all right but the second yields a result of zero. This is because the VAL function aborts whenever is encounters a non-number character. Note that 'number' characters include the +- and E characters provided they are in their expected position.

STR$ is the opposite of VAL; it converts numerical values to strings. Let's look at one application of this function to RIGHT JUSTIFY a column of numbers. Type in the following program and give it 10 values to display. Remember to include some MINUS figures and some with 1,2, and 3 decimal places:

```
10 PRINT "Give me 10 dollars and cents figures"
20 FOR x= 1 TO 10: INPUT d(x): NEXT
30 FOR x= 1 TO 10
40 d=d(x)
50 GOSUB 1000: REM print it right justified
60 NEXT x
70 END
999 REM routine to print d (right justified)
1000 d$="$": if d<0 then d$="-$": REM set the prefix string
1010 d=ABS(d): REM strip minus sign if any
1020 t$=d$+STR$(INT(d)): REM get whole dollar value
1030 PRINT SPC(20-LEN(t$));t$;: REM print dollars
1040 c=d-int(d)+.00501: REM get the pennies and round up
1050 IF c<.01 then PRINT ".00":RETURN: REM abort if none
1060 PRINT LEFT$(STR$(c)+"00",3): REM add in decimal and pennies
1070 RETURN
```

The first 2 lines of the subroutine decide whether to start with $ or -$ based on the sign of the 'd' variable. 'd' is then converted to a positive number. In order to right justify around the decimal point, the WHOLE DOLLAR portion of 'd' is extracted via the INT function and converted to a string via the STR$ function. When this is added to the d$ which was defined in line 1000, we have the total number of characters to be printed left of the decimal. Line 1030 accomplishes the right justification by printing spaces equal to the difference between 20 (our pivot point) and the length of the string. The dollar value is printed. Line 1040 figures out what pennies

remain and line 1050 uses LEFT$ (covered below) to print exactly 3 figures which may or may not be zero.

LEFT$ extracts a portion of a string from it's start position to a specified length. The syntax is LEFT$(string,length) where string can be any valid string expression, (even string addition is allowed) and length in an integer from 1 to the length of the string. Note that any string parameter which exceeds the length of the string does not generate an error, it simply returns the maximum possible value based on string length.

RIGHT$ extracts a portion of a string from it's right side (or end). It uses the same syntax as LEFT$. Consider the following example of LEFT and RIGHT.

```
10 INPUT w$
20 y=LEN(w$)
30 for x= 1 to y-1
40 PRINT LEFT$(w$,x);" + ";RIGHT$(w$,y-x)
50 NEXT x
```

MID$ is much more versatile in that it allows the extraction of any part of a string whether beginning, middle, or end. The syntax is
MID$(string,start,length) where string is any valid string expression, start is the integer position to start and length is the number of characters to extract. If the length is not provided, MID$ will extract from current position to the end-of-string. This is an undocumented and very powerful feature of string operators. Consider the example above. In order to print the second half, it was necessary to subtract the start position from the total length of the string. Using MID$, this operation is not necessary, nor is it necessary to even know the length of the string. Consider the following:

```
10 INPUT w$
30 for x= 1 to 10
40 PRINT LEFT$(w$,x);" + ";MID$(w$,x+1)
50 NEXT x
```

If the string is 10 or shorter, this routine will behave in an identical fashion to the routine using RIGHT$. Note, however, the simpler arithmetic for the second half: start at x+1 for the rest of the string.

When dealing from a 'deck', programmers usually turn a 'card' ON or OFF when dealing. When a random card is selected, it's availability is checked prior to selection. While this works reasonably well with small arrays, larger arrays (even 52) often cause problems when dealing those last few cards: we just can't seem to randomly hit them. Consider the following example which uses string manipulation to solve the problem. Our first assumption is that cards are numbered from 1 to 52. The cards from 1 to 13 are clubs (for example), 14 to 26 are diamonds, etc.

```
99 REM initialize the deck
100 d$=""
110 FOR x=1 to 52
120 d$=d$+chr$(x): REM put each card in line
130 NEXT x
140 RETURN
199 REM pick out one card from the deck
200 l=len(d$): REM get the remaining length
210 r=int(rnd(1)*l+1): REM get a random position in the deck
220 c=asc(mid$,d$,r,1): REM get the card value
230 a$="":if r<>1 then a$=left$(d$,x-1): REM get first half
240 b$="":if r<>l then b$=mid$(d$,x+1): REM get second half
240 d$=a$+b$: REM crunch the deck
250 RETURN
```

We have 2 subroutines. The first at 100 initializes the deck. Although this is not shuffling, picking out cards at random within the deck will have the same effect. The second routine at 200 figures out the number of cards remaining in the deck and picks out a random number in that range. Variable "c" contains the value of the selected card from 1 to 52. Lines 230 and 240 cut the deck in 2 sections, the first before the chosen card and the rest after it. Note the protection which MUST be included in the event that the chosen card was the first or the last card. Finally, the 2 halves are added together to form the REMAINING deck. If you have ever dealt

cards at random, try this routine. You will be pleased to see that it runs much faster than other methods, particularly when the deck is nearly empty. You always get a hit instead of trying to pick cards which have already been used until you find a good one.

ROUTINE ADDRESSES

LEFT$ executes from 10508(290C) to 10526(291E). It calculates string length and cut point and calls the cut routine at 10616(2978).

RIGHT$ executes from 10529(2921) to 10550(2936). It also calculates the start point and length to extract and calls the cut routine.

MID$ executes from 10553(2939) to 10615(2977) with the actual entry point at 10563(2943). It gets the start point and checks if a length was supplied. If there is none, the remaining length is used. It eventually falls through to the cut routine.

LEFT RIGHT and MID usa a complex compare routine from 10464(28E0) to 10507(290B). It checks the requested length against the length of the string.

LEN executes at 10454(28D6). It simply extracts the string length byte from the floating point accumulator (which is used top store string pointers).

STR$ executes from 10411(28AB) to 10453(28D5). It converts the numerical value to ASCII as if it was to be printed on the screen, then creates a string of the appropriate length.

ASC executes from 10351(286F) to 10368(2880). It gets the pointer to the string and extracts the first character. It then dumps the value into the floating point accumulator for use by the rest of the logical operation.

CHR$ executes from 10371(2883) to 10410(28AA). It creates a string of length 1 and fills in the value of the string with the number supplied.

VAL executes from 10309(2845) to 10348(286C).

MATHEMATICS FUNCTIONS

In the several previous articles on SMARTBASIC, I have mentioned variable commands. Although most of them are mathematical functions, a few are not; notably FRE and USR. Variable commands are those commands which pass a parameter within brackets: eg INT(123.45). The parameter is evaluated by the function in order to determine the result. Presumably to save on interpretation and parsing code, the designers of SMARTBASIC adopted a complicated technique which dynamically relocates these variable commands based on the LOMEM setting. Each of the variables is defined as an array and the array simply point back to the execution routine for each function.

When you are playing around with memory and accidentally write where you should not, the variable commands are invariably the first ones to suffer. When they start misbehaving, the best thing to do is reboot.

In this article, we will briefly cover the use of arithmetic and algebraic functions. I will not attempt to describe the calculation method, for even if I understood it completely, it would take several pages to explain. The purpose of this article is to remind you that these functions are there and clarify their use as required.

INT does just what you might expect; it extracts the integer value of a real variable. Since it truncates rather than round off, statistical calculations will be more precise if you use INT(x+.5). You will also find that certain numbers truncate in a strange fashion. I have never noted the exact numbers, but the floating point has difficulty handling numbers like .001. For this reason, I often use INT(x+.50001). This helps to avoid those nasty INTEGER values which wind up being 37.9999997.

ABS takes the absolute value of a number by removing its sign thus ABS(-12) will yield 12. You can use ABS to make a number negative with something like -1*ABS(x).

SGN will report on the sign of a variable. SGN will return 0 if the variable is 0, -1 for negative values, and +1 for positive values. SGN can be used in conjunction with ON GOTO in the following fashion:

```
999 REM make decision on sign of x
1000 sx=SGN(x)+2:  REM make result 1,2,3
1010 ON sx GOTO 2000,3000,4000
2000 REM handle negative
3000 REM handle zero
4000 REM handle positive
```

LOG takes the natural LOG (base e) of a number. If you are curious, the value of e is 2.718281828....You can come close to this value by asking SMARTBASIC for the LOG(10). If you want to take base 10 LOGS, just divide the LOG value by LOG(10):

```
999 REM subroutine to take base 10 log of x
1000 y=LOG(x)/LOG(10)
1010 RETURN
```

EXP is the complementary function which raises e to the power of the argument. Thus EXP(2) is the same as e^2. This function is redundant for powers of 10 since you can use 10^2 or 1.0E+2. It would be tedious, however to write 2.718281828^2.

SQR extracts the square root. Thus the Pythagorean theorem would be calculated by hyp = SQR(s1^2 + s2^2). The square root can also be expressed with hyp = (s1^2 + s2^2) ^ (1/2), but SQR is more convenient.

Before discussing the TRIG functions, a bit about RADIANS. Computers insist on working with radians rather than degrees. If you remember your high school trigonometry, there are 'pi' radians in 180 degrees or about 57.3 degrees per radian. 'pi' (despite what textbooks might say) has the value 3.141592657 and you can define RAD=180/3.141592657 to use as a conversion from degrees to radians. This will become clearer in a moment.

SIN takes the sine of the specified radian. If you would rather work in degrees, use something like SIN(45/RAD) to evaluate the sine of 45 degrees.

COS takes the cosine of the specified radian. Again, if you remember you high school math, COS(x)=SIN(90-x). Thus COS(45) should be the same as SIN(45). Define RAD as outlined above and print SIN(45/RAD) and COS(45/RAD). If you change the value of 'pi' in the equation to a different value like 3.141592655, you will see that the values are not the same. Thus the value given above is the CORRECT one for working with ADAM's floating point accumulator.

TAN takes the tangent and ATN takes the arc-tangent; the latter function is difficult to calculate manually.

All other TRIG functions can be evaluated using the 4 functions above; you just have to remember how it's done. I must admit I have forgotten.

ROUTINE ADDRESSES

INT executes at 10672(29B0). It verifies that the number is in floating point format. It then checks if the number is less than 1 in which case 0 is returned. It then juggles the number around to drop the decimal portion.

ABS executes at 2276(08E4). It simply resets the sign bit in the floating point accumulator. This is why ABS cannot be used with INTEGER variables.

SGN executes at 2285(08ED). It starts by checking the value of the exponent in the FPA. If zero it simply returns which yields a zero value. It then sets the value in the FPA to +1 or -1 depending on the original sign of the number.

LOG executes at 3604(0E14). It checks for zero and negative values and then calls a power series calculator to do a recursive calculation.

EXP executes at 3816(0EE8). It uses a power series calculator to approximate the required value.

SQR executes at 3678(0E5E). It does the calculation the way we learned it when we learned about logs. It takes the LOG of the value, divides it by 2 and recalculates the exponent. This is like doing e^(log(x)/2).

SIN executes at 3954(0F72). This is the workhorse which uses a power series calculator to approximate the required value.

COS executes at 3946(0F6A). It calculates the value from the formula COS(x)=SIN(x+pi/2) in radians, of course.

TAN executes at 3912(0F48). It calculates the value from the formula TAN=SIN/COS.

ATN executes at 4180(1054). It also uses a power series calculator.

Following is a list of routines used by the MATH functions. They are very complex and should likely not be fiddled with. The addresses are included for information purposes only.

4156-4179 POWER SERIES CALCULATOR
It takes a value from a table pointed to by HL. Copies FPA1 to FPA2, multiplies the first value at (HL) by FPA1 and multiplies the original result by the next table value. It is called by the controlling routine as long as there are values in the table.

4255-4269 ADD PI FRACTIONS
Depending on the entry point, these routines add or subtract pi/2 or pi/4 to the current value in the floating point accumulator.

4270-4355 POWER SERIES CALCULATOR
This one basically multiplies the 2 numbers in the FPA's, then multiplies each component by values from the power table and finally adds the 2 halves together.

4497-4695 CONSTANTS
This is a set of values used by the various math routines when calling the power series calculators. Some are in floating point and others in integer format. You will find values corresponding to LOG(2) 1/LOG(2) pi/2 2/pi pi/4 1/11 1/9 1/7 1/5 1/3 but NOT pi or e.

RANDOM NUMBERS

I have written on several occasions about random numbers. I will probably continue to do so until such time as we have exhausted all the questions, gripes, and hacks. This expose will explain how and why RANDOM works, and help you get the most out of it.

The RND function returns a random number between 0 and 1. Although the value is never 1, it is possible for it to equal exactly 0. There are 3 ways to ask for a random number:

RND(1) or RND(2), or any positive number, will extract the NEXT random number from the generator. The argument value is unimportant.

RND(0) will restore the previously generated random number. This might be useful if your program FORGETS what the last random number was and you want to double check its value without affecting anything else

RND(-x) will reset the random seed to a particular value based on the supplied number. This can be useful in GAME debug situations where you may want to recreate an exact condition.

Now what to do with random numbers? What use is a number between 0 and 1. Well quite simply, you multiply it by the number of choices you have to make. If you want to randomly decide whether to go North South East or West, use something like: move=INT(RND(1)*4+1). Note that we multiply by 4 (the number of choices) and add 1 before taking the INT. This will give us a random INTEGER which is 1, 2, 3, or 4. This result can be used with an ON GOTO statement to branch to the correct routine.

Random number generation on the ADAM is accomplished by a complex routine at 4696(1258). It either loads the floating point accumulator with the 4-byte CURRENT random seed, or with 4 bytes representing the use-supplied value if RND(-x) was used. It then points to 4 constants at address 4552(11C8). These values are 45, 230, 64, and 187. You can change those if you wish but they will likely cause your random number generator to fail. Each of these numbers is multiplied in turn to the current value in the FPA. After each recursion, one of the 4 CURRENT RANDOM bytes is updated with the OVERFLOW of the calculation. The resulting value in the floating point

accumulator is then reduced to a number between 0 and 1 to return to the caller. If all that sounds confusing, it is! But there is more...

NEW and RUN insist on re-initializing the random seed by copying the 4 static bytes FB 40 D2 92 into the random generator at address 16190 (3F3E) and 16192 (3F40). The routine that accomplishes this task is found from 11907 (2E83) to 11918 (2E8E). This is supposed to prevent the random number generator from breaking down but it has the effect of generating the same random numbers every time a program is run. You may have seen some random routines which try to overcome this problem. While there are many approaches, the following is my favourite to generate a random seed in simplicity and safety -- it is a hard one to CHEAT:

```
100 REM ask for instructions and set random seed
110 REM can be included at any program start
120 REM Instructions or help question is a convenient way
130 REM of getting a key press
140 REM
150 PRINT "Do You Want Instructions (Y/N)?"
160 p=PEEK(64885): REM record current key press value
165 REM stay here until a different key is pressed
170 if p=PEEK(64885) then r=RND(1):goto 170
180 p=PEEK(64885):REM get current key
190 REM now check for y Y n N
200 REM if none of the above branch back to 170 until different
```

While not fool proof, this method will give the hackers a hard time. But look at it this way: If I play a game it's no fun if it is the same all the time. Those who want to play THE SAME GAME will find a way so don't bother with them.

But why should BASIC reset the seed and force me to do all this extra work? NO REASON. As a matter of fact, you can POKE zeroes into addresses 11907 to 11918 to disable the re-setting of the random seed. I have experienced no ill effects from this approach.

Furthermore, for those using my zero page clock, you can make your random generator use the Hours, Minutes, Seconds, and Jiffies as a random seed with the following:

```
10 DATA 42,84,0,34,62,63,42,86,0,34,64,63
20 FOR x=11907 to 11918
30 READ y
40 POKE x,y
50 NEXT
```

Add this routine to your HELLO file and you won't have to worry about randoms any more. Remember, however, that the clock is turned off when in GRAPHICS modes. A game that terminates in graphics mode will restart with the same random seed if RUN is typed from the graphics mode. Accordingly, those games should end with a TEXT command to restart the clock.

With special thanks to Bob Currie EAUG (Edmonton) and the ANN network which published his article on random numbers, I have another addition to the randomness of random numbers. Following is an extract from his article:

"If we scan all of the accessible memory locations in the ADAM, we find that there are three addresses at which one does not always get the same number. At 17003(dec) we get a zero seven times out of ten and a one three times out of ten. At 65220(dec) we get a 4 seven times out of ten and a 140 three times out of ten. At 17011(dec) we get the numbers from 1 to 12 with a pretty much even chance of getting any one of them."

Bob goes on to say that you can add the 3 values at these addresses to form the basis for a RND(-x) routine. I have come upon a more ingenious method. Firstly, adding the numbers together gives results ranging from 5 to 17 or 141 to 153 or 26 unique combinations. If we instead use the value at 17003 as a multiplier, we can increase the range to 48 unique combinations:

```
r=PEEK(65220)+PEEK(17011)+12*PEEK(17003)
```

The first half of the equation yields 5 to 16 or 141 to 152. Adding 12 times the value at 17003 expands this range to 5 to 28 or 141 to 164. But now what do we do with this number? Just POKE it into one of the RANDOM

SEED values prior to scanning the keyboard:

```
100 REM ask for instructions and set random seed
110 REM can be included at any program start
120 REM Instructions or help question is a convenient way
130 REM of getting a key press
140 REM
145 POKE 16191,PEEK(65220)+PEEK(17011)+12*PEEK(17003)
150 PRINT "Do You Want Instructions (Y/N)?"
160 p=PEEK(64885):  REM record current key press value
165 REM stay here until a different key is pressed
170 if p=PEEK(64885) then r=RND(1):goto 170
180 p=PEEK(64885):REM get current key
190 REM now check for y Y n N
200 REM if none of the above branch back to 170 until different
```

LOGICALS

Logicals are perhaps the most misunderstood portion of BASIC programming. Yet they can be one of the most powerful programming tools. Not only can logicals make programs shorter, they can make them run much faster.

What is a logical? It is an instruction which uses the result of an operation to make a decision. One of the more common examples is conditional branching:

```
100IF x>0 GOTO 500
```

When the computer encounters the "x>0" it checks whether this is true or false. If it is true, a LOGICAL TRUE (1) is assigned, otherwise a 0 is assigned. If the result of the logical is true, the rest of the command line is executed.

The AND logical will yield a TRUE result only if BOTH supplied equations are true. The OR logical will be true if either equation is true. Thus to make a jump if either player has a score of 100, you would say:

```
100IF p1=100 OR p2=100 GOTO 500
```

To jump if player 1 has 100 AND the number of turns is 20, you would say:

```
100IF p1=100 AND t=20 GOTO 500
```

The NOT logical REVERSES the value of a variable. If it was non-zero it becomes zero; if it was zero it becomes 1.

```
100x=100:PRINT x
110x=NOT x:PRINT x
120x=NOT x:PRINT x
```

would yield 100,0,1 respectively. How useful is this? Let's take a simple game example where the computer and player take turns playing first.

```
100x=NOT x:IF x GOTO 500:REM make computer move first.
```

Note also the use of "IF x" which reads "if x not equal to zero". Every second time this statement is executed, x will be non-zero and the jump will be made.

Other logicals include = <>(not equal) > < >= => <= =<. Note that the last 2 pairs are equivalent and mean greater or equal and less than or equal. These logicals are often used in programs but they can usually be made more effective by careful planning:

```
100IF x=2 GOTO 110
105y=y-1
106GOTO 120
```

```
110y=y+1
115z=z/2
120more program

100IF x=2 then y=y+1:z=z/2:goto 120
110y=y-1
120more program
```

Both routines accomplish the same task but the second works much faster sine there is less jumping around. Remember that when a logical is false, the rest of the line is discarded, not just the rest of the statement. IF you look for the condition which requires the least amount of calculations, it can be put on the first line and jump over the next. Therefore in the example above, a better way would be:

```
100IF x<>2 then y=y-1:GOTO 120
110y=y+1:z=z/2
120more program
```

When dealing with integer values, avoid the >= and <= logicals when possible. For example:

```
IF x<=5 GOTOtakes more time than
IF x<6 GOTO
```

If it is possible for x to have a value of 5.1 then the second equation will not have the desired effect. BUT

IF x<5.00001 will work, and faster.

Now for something a bit more complicated. Let's say we have a menu selection of "Edit Sort File Print Quit" that we want to evaluate using the single keystrokes "esfpq":

```
100GET q$
110IF q$="e" GOTO 500
120IF q$="s" GOTO 600
130IF q$="f" GOTO 700
140IF q$="p" GOTO 800
150IF q$="q" GOTO 900
160GOTO 100:REM try again
```

This can become quite tedious. Remember that logical TRUE is 1 and that FALSE is 0 and look at the following:

```
100GET q$
120q=1+(q$="e")+2*(q$="s")+3*(q$="f")+4*(q$="p")+5*(q$="q")
1300N q GOTO 100,500,600,700,800,900
```

It may look a bit messy at first, but it works much better and saves program lines. Firstly q will always equal at least 1 which will be compatible with ON GOTO. Secondly, only one of the values within parentheses will be true and q will be assigned values 2,3,4,5,6 depending on the value of q$. If an incorrect entry is made, q will be 1 and the ON GOTO will send it back to get another key press. Note that neither of the 2 routines above handles uppercase characters. This could be done in either one by using something like q$="e" OR q$="E".

Remember the joystick routines we illustrated earlier? It talked about reading the joystick for NSEW values:

```
              NORTH
                1
                |
 NORTH WEST - 9 | 3 - NORTH EAST
               ¥|/
     WEST - 8<- 0 ->2 - EAST
               /|¥
SOUTH WEST - 12 | 6 - SOUTH EAST
                |
```

4
SOUTH

From the diagram, you can see that NORTH EAST SOUTH WEST are assigned the values 1 2 4 8 and that the intermediate points are the sum of the corresponding values. The first thing we did was read the PDL value into variable 'p'

```
100 p=PDL(5)
```

The next task was to decide whether to increment or decrement the x or y coordinate. We will did this using complex logical equations:

```
110 y=y+(p=4 or p=6 or p=12)
120 y=y-(p=1 or p=3 or p=9)
130 x=x+(p=2 or p=3 or p=6)
140 x=x-(p>7)
```

This increments/decrements x and/or y by one based on the chosen direction. This can further be improved by setting an upper and lower limit (game board border):

```
110 y=y+(p=4 or p=6 or p=12)*(y<10)
120 y=y-(p=1 or p=3 or p=9) *(y>1)
130 x=x+(p=2 or p=3 or p=6) *(x<10
140 x=x-(p>7)*(x>1)
```

Thus y is only incremented if the y value was smaller than 10. If 10 or more, the second logical in line 110 is false and returns a 0 which multiplied by the preceding value will always be 0.

Although SMARTBASIC's logicals set is not the most extensive (no ELSE WHILE WHEN, etc) the available ones will let you make any decision or calculation. Effective use of logicals can reward you with program speed in a reduced size.

There is not much to be done with the routines which evaluate logicals. For those who are curious, following are the execution addresses:

```
AND54961578
OR55131589
NOT57051649
=561715F1
<>558715D3
>554115A5
<55271597
>=557315C5
<=555715B5
```

5939 (1733) gets an equation from the program line. If there is an operator, it evaluates the result.

5760 (1680) gets an operator (if any) and extracts the execution address shown above.

FILE COMMANDS

This article will cover the elementary file related commands in SMARTBASIC. The special commands related to data files will be dealt with in subsequent articles. There are 9 elementary commands

which allow you to perform housekeeping functions:

CATALOG is perhaps the most often used. It reports the files on a medium. Unlike the DIR command in CP/M, you cannot specify which files you are looking for, the catalog command always reports all files. The catalog command is usually the vehicle to change the DEFAULT drive. If you are currently working on tape drive 1 and wish to access disk drive 1, you would type:
CATALOG,D5

If you are wondering why D5, Coleco had originally intended to supply 4 tape drives for the ADAM, numbered 1 to 4. This is why the disk drives are 5 and 6. Note that it is not essential to supply the comma after the CATALOG command. Experienced programmers find that typing:

CATALOG D5

is much faster and saves keystroke errors.

RENAME is used to change the name of a file. It uses straightforward syntax:

RENAME oldname,newname

If the destination file name exists, there is disk activity but the file is not renamed. Neither do you get an error message! This is yet another of the bugs in SMARTBASIC, it fails to check for and report the error returned by the EOS. Because of the complexity of the routines involved with rename, this would be a difficult problem to correct. Note that you can also specify the drive on which the file is located with a ",d5" for example for disk 1.

RECOVER is used to recover a backup file "a" or "h" files. If you wind up making a program worse by trying to make it better, you may wish to go back to the previous version. In this case, you would delete the current "A" file and type:
RECOVER filename

As with the rename command, nothing will happen if you try to recover a file when the main "A" file still exists. There is a further bug in SMARTBASIC which prevents the recovery of "H" files. This is because the recover routine tries to change the file type from "h" to "h" thereby making no changes. To fix this, you simply POKE 20619,ASC("H") that is decimal 72. In order for recover to work, there must be no file of the requested name with a file type of "A" or "H". You can use the RECOVER routine to create you own special file types. Simply POKE the CURRENT FILE TYPE in 20614 and the desired file type in 20619. This will protect your files from undesired access. To restore the file to a standard type, simply poke its special file type in 20614 and the correct file type in 20619. Recover also accepts a drive name.

The DELETE command is used to remove a file from the directory. Be careful not to confuse the DELETE command with the DEL command which is used to delete a range of program lines. The only file types which can be deleted are "A" and "H" files. "a" and "h" files must be recovered before being deleted. This was COLECO's idea of protecting the clumsy programmer who went around deleting his programs. This tedium only serves to frustrate as it creates an accumulation of files. Once a program is finalized and you want to get rid of the "a" file you left behind, you have to rename the final to something else, recover the "a" file, delete that file, and rename your program. That's why we use programs like FILEMANAGER to clean up disks by removing BACKUP files. The delete command also accepts a drive name. Note that if a file is LOCKED (see below), the delete command will abort with a FILE NOT FOUND error.

LOCK is used to prevent the accidental deletion of a file. It has a simple syntax:
LOCK filename

When you issue a CATALOG command, you will see an asterisk beside the file name confirming its locked

status. To unlock a file, simply type:

UNLOCK filename

LOAD is used to load in "A" type programs. The load process is quite slow since BASIC must interpret all the commands and store them in tokenized form in memory. Once the prompt returns, you can LIST or RUN the program. The LOAD command allows you to fetch a file from a different drive by suing the following:
LOAD filename,d5

for example, to load a file from disk drive 1. Note that the RUN command can be used to load and run a file in one operation:

RUN filename

This command, however, does not maintain program pointers. To illustrate, type and save the following program:

10PRINT "START"
20FOR x=1 to 1000
30NEXT X

LOAD and RUN the program, wait a second and press CONTROL-C followed by PRINT x. You will see the value of x at the point of interrupt. Now type RUN programname and press CONTROL-C after the START appears on the screen. When you type PRINT x, you get a value of 0 which is obviously wrong. When debugging programs, you should always use LOAD and RUN.

This brings up an interesting bug in RUN which may have been encountered by other clumsy typists like myself. LOAD in a program and type:

RUN]<CR>

Note that the ']' is close to the return key and that is easy to accidentally hit both simultaneously. You get a RANGE ERROR which does not appear to be a problem. But try and list the program....it is gone! This is because when RUN is followed by another character, it is presumed to be a file name. Memory is cleared and BASIC attempts to load in the program. If the syntax check was done first, this problem would not arise.

SAVE will save your "A" type programs to tape or disk. Each save operation renames the current file to "a", and then saves the new program as a type "A". If there was already a type "a" file, it will be deleted. Programmer's manuals tell you to save programs frequently when developing them to protect against crashes. Let's say you save your test program about every 50 lines. After 5 saves, you will have the following directory entries:

test a1K (deleted)
test a1K (deleted)
test a2K (deleted)
test a3K
test A4K

All these entries waste valuable directory space. The new saves cannot be re-written in the deleted entry space since the program is now bigger. This is another illustration of the need for a good utility like FILEMANAGER to recover deleted file space.

INIT is used to wipe a disk or tape clean by blanking out the directory. The INIT routine allows you to assign a volume name to your disks. If you have a disk cataloguing utility like EOS INDEXER,

volume names are essential. Use the following:
INIT volumename

where volumename can be up to 12 characters. The INIT sequence, however was designed before knowing the size of disk drives. It has no direct provisions for specifying the 160K or 320K size for disks. See the technical section below for the specific addresses for volume size and directory size.

All of these file commands can be given in CONTROL-D format, that is within programs. This requires a special syntax:

commands must start with a CONTROL-D;
the CONTROL-D must be the first character on a line;
the command must be 23 characters or less.

There are several ways to print the CONTROL-D at the start of a command. One method is to assign a string variable the value of CONTROL-D:

```
10d$=CHR$(4)
20PRINT d$;"CATALOG"
30PRINT d$+"CATALOG"
40PRINT CHR$(4);"CATALOG"
50PRINT CHR$(4)+"CATALOG"
```

All of the above are valid forms of the CATALOG command within a program. They all have potential problems when dealing with large programs: sometimes string variables get garbled and other times, the MATH functions behave erratically. It can be crucial when dealing with data files to properly save the file regardless of these problems. Hence my preferred approach. Type

```
10PRINT "
```

after the quote, hold the CONTROL key and press D. Even though nothing appears on the screen, the CONTROL-D is there. Continue without any spaces and write

```
CATALOG"
```

Now list line 10 and you will see a heart after the quote. This confirms that the CONTROL-D has been entered on the command line. Occasionally, this may present a problem when several commands are written on the same line. When listing a program, if the CONTROL-D winds up at the start of a line SMARTBASIC will complain with a syntax error and the rest of the line will be lost. Don't panic! type:

```
PRINT, : LIST linenumber
```

This will offset the line by 1/2 screen and you can either retype the line or scroll through it inserting a space somewhere to make sure the CONTROL-D does not appear at the beginning of a line.

The 23 character limit for the CONTROL-D buffer may appear quite adequate, but consider the following:

```
RENAME oldfiles,newfiles
BSAVE filename,a1000,l12345
```

These commands are 24 and 27 characters respectively and will result in a CONTROL BUFFER OVERFLOW error. The CONTROL-D buffer can be moved to a larger area with the following POKES:

POKE 19566,167 POKE 19567,64(16551)
POKE 19585,132 POKE 19586,64(16516)
POKE 19459,131 POKE 19460,64(16515)
POKE 17040,132 POKE 17041,64(16516)

this gives you a 35 character buffer which should be more than adequate.

ROUTINE ADDRESSES

3 addresses will be given for each routine. The first is the controlling routine for regular commands, the second the controlling routine for the CONTROL-D command, and the last is the main routine which performs the major task. each of the controlling performs syntax checks and jumps to an error handler after the routine execution. This complicated setup is required since errors are handled differently in direct mode compared to commands under program control.

REGULARCONTROL-DROUTINE
CATALOG19971 (4E03)19693 (4CED)21144 (5298)
The routine starts by reading block 1 of the selected medium. It then prints the volume name (5353). It proceeds to read the 37 next entries and checks them for the BLOCKS LEFT attribute; note that BLOCKS LEFT is not required, just the attribute. It then checks for system files and skips if so; to defeat the system file check, POKE 21240,0. The next check is for deleted files; to show deleted files POKE 21243,24. When a deleted file is encountered, the size of the file is computed and added into a buffer for the blocks left calculation. The routine at 5368 prints a file name in the following fashion: Check the attributes for locked and print a * if so; decode the file name and print the attribute; print the file size; and then the file name. To change the LOCKED FILE indicator, POKE the ASCII value of the character to use in 21405. After 1 K of directory is read, another 10 bytes are flushed out (those are the leftovers-->1024-39*26=1014. It then proceeds to read in the next K and read in another 39 files. This process continues until a BLOCKS LEFT attribute is encountered. The directory routine pays no attention to the DIRECTORY SIZE entry and could go on forever if BLOCKS LEFT is not found. Once it is found, the wrap up routine at 5327 takes over. It takes the blocks left and adds in the amount of deleted file space to report blocks free. Note that deleted file space is not always usable and that this method of calculation although correct may be misleading. If you want to know the number of blocks available after BLOCKS LEFT, POKE 21297,0.

RENAME20001 (4E21)19712 (4D00)20469 (4FF5)
This routine starts by checking the source and destination file name for validity. It then tries to change an "A" file type to the new name. If none is found, it tries an "H" file type. If neither is found, it jumps to the error processor. One interesting thing you can do is rename an 'A' file to any other file type you wish, or vice versa. To do this, POKE the ASCII value of the SOURCE file name in 20482 and the ASCII value of the destination file name in 20487.

RECOVER20114 (4E92)19875 (4DA3)20532 (5034)
This routine is very similar to rename. It checks the validity of the file name and the drive and then checks for the existence of an "A" or "H" file of the same name. It then tries to RENAME and "a" file to "A"; if that does not work it tries "h".

DELETE19983 (4E0F)19701 (4CF5)20426 (4FCA)
This routine also starts by checking the file name and drive. It tries to delete an "A" file type first. If none is found, it tries an "H" file type. If it should happen that you have an "A" and "H" file of the same name, the "A" will always be deleted first. If neither file is found, the error handler takes over.

LOCK20008 (4E28)19719 (4D07)20641 (50A1)
UNLOCK20015 (4E2F)19726 (4D0E)20640 (50A0)
These routines are shared and use different entry point to set the carry flag if LOCK is selected.

They both start with a check of the file name and drive and look for an "A" or "H" file matching the request. If neither is found, the error handler takes over. The attribute byte is read in and shifted over and then the PERMANENT PROTECT bit is checked; if so, nothing is changed. The carry flag is used to shift the LOCK UNLOCK status into the attribute byte and the directory is updated. To defeat the PERMANENT PROTECT check, POKE 20705,0.

LOAD20064 (4E60)19775 (4D3F)23976 (5DA8)
The load routine uses a special trick to set the READ FROM FILE flag to replace normal keyboard input. It then jumps to the NEW routine which clears memory and reads input from the file as if it had been read from the keyboard. When the end of file is reached, normal keyboard input is resumed.

SAVE20071 (4E67)19782 (4D46)23813 (5D05)
This routine works in a similar fashion to LOAD, but it first checks the validity of the file name and drive. It creates a temporary file name $$$$1 or $$$$2. It then calls the LIST routine repeatedly to list all the file to TAPE instead of SCREEN. Once the end of program is reached, it closes the file and renames it to the requested name. If you want your SAVEd file to have the same appearance as when it is listed to the screen, POKE zeroes into addresses 24100, 24101, 24102. The default values are 50, 20, 63.

INIT20096 (4E80)19857 (4D91)25267 (62B3)
This routine starts again by checking the volume name and drive. It then checks for the existence of a system file called BASICPGM on the disk and aborts quietly if that name is found. This technique prevents you from accidentally INITing your SMARTBASIC tape. It then fills in all the required values for directory size, drive size, etc. Here's where you can make changes to correctly initialize TAPES or DISKS

ITEMADDRESSVALUES
Drive Size25305,25306number of K in medium (lo-hi)
255 for tapes
159 for disks
319 for DS disks

Directory 25308number of K reserved for DIR
Size

Next, the BOOT directory entry is written to the medium. The next step is to WRITE a JP to SMARTWRITER in block 0 so nothing strange happens if the reset switch is pulled with this medium in the drive.

Note that the EOS INIT routine which is called by SMARTBASIC does most of the work. Because of disk buffering, when the JP SMARTWRITER instruction is written to block 0. the rest of the first K of directory is also copied to block 0. Thus to UNINIT a 1K medium which has accidentally been killed, pull out the disk editor and copy block 0 to block 1. The only other thing to do is fix the first three characters of the volume name.

READING DATA FILES

This chapter will cover the mechanics of data file manipulation in SMARTBASIC. We will also touch on some examples of data file structure and some productivity hints. The first part will only deal with the basic mechanics of data file access. Article 2 will further refine the strategies.

Before examining the commands for accessing data files, a bit about buffers. SMARTBASIC has 3 1K buffers that are used for a variety of purposes. These are located at D400, D800, and DC00. The last

one resides just below the start of the EOS which starts at E000. The first buffer is used to read the directory of the medium. Even on directories greater than 1K, this is the only buffer used for directories. The other 2 can be used for file access. They are used, for example, when loading or saving a basic program. They can also be used to open and read data files.

BASIC automatically assigns one or the other to the file being accessed. You can have 2 files open simultaneously but they must be on the same drive. If you try to open 2 files on 2 different mediums, you will get a NO BUFFERS AVAILABLE error since the directory buffer is currently in use for the first file. This means that you cannot use DATA FILE READ commands to transfer a file from one medium to another unless you first close the input file before opening the output file.

The file commands illustrated in this article, with the exception of MON and NOMON, must be used in CONTROL-D format as was outlined in the previous chapter. Accordingly, the sample programs illustrating the commands will show a ^D in the print statements to signify the CONTROL-D character.

The OPEN command is used to open an existing file or create a new one. IF the file does not exist it is automatically created. This in itself can create problems. If you issue the OPEN command when you have the wrong disk in the drive, you will wind up creating an empty file. Your controlling program may behave strangely when it discovers that there is no data.

```
10PRINT "^DOPEN filename"
```

Alternately, the controlling program may prompt the user for a file name and pass it to the OPEN command:

```
10INPUT "File to read ";f$
20PRINT "^DOPEN ";f$
```

Note that there is a space after OPEN; it is required so BASIC can parse the OPEN command.

When you first create a DATA file the next thing you want to do is write data to it. The open command only makes the file available for access. You must tell BASIC that you want to write to it.

```
10INPUT "File to read ";f$
20PRINT "^DOPEN ";f$
30PRINT "^DWRITE ";f$
```

Since you may have more than one file open, the WRITE command must also include the file name. If you try to WRITE to a file that has not been OPENed, BASIC will complain. After the WRITE command has been issued, all PRINT statements will go to the file; nothing goes to the screen. Thus it is impossible to send informational messages to the screen while the WRITE command is in progress. All PRINT commands will be formatted in the file the same way that they would have appeared on the screen. The conclusion of a PRINT statement send a carriage return to the file to specify the end of the line. Thus, to send more than one STRING to the data file, you must use the ";" or "," characters:

```
10INPUT "File to read ";f$
20PRINT "^DOPEN ";f$
30PRINT "^DWRITE ";f$
40PRINT "My name is Guy Cousineau"
50PRINT "This data is being sent to ";f$
60PRINT 1, 2/3, 5.18, 564*765
```

Once you are finished writing to a file, you must CLOSE it to record the file on the medium, and to restore normal screen output:

```
10INPUT "File to read ";f$
```

```
20PRINT "^DOPEN ";f$
30PRINT "^DWRITE ";f$
40PRINT "My name is Guy Cousineau"
50PRINT "This data is being sent to ";f$
60PRINT 1, 2/3, 5.18, 564*765
70PRINT "^DCLOSE ";f$
```

Note again the space after CLOSE and that the file name must be specified. Now we have a program that will create a data file. Run this program, BOOT SMARTWRITER, and have a look at the file. It should look like:

```
My name is Guy Cousineau
This data is being sent to testfile
1   .66666666  5.18  431460
```

Because the print command on line 60 was ALL-IN-ONE-LINE, that is exactly the way it will appear on the screen. DATA files may have lengths up to 128 characters, but if you intend to read them from SMARTWRITER, you should limit line length to 80 characters.

Now return to BASIC and run the test program again using the same file name. What do you think will happen? Go back to SMARTWRITER and look at the data file again; it looks just the same as before. That is because the WRITE command always writes to the start of the file. This poses a serious problem. If you supply the wrong name for your data file, and it is a file that already exists, you will erase any previous information on that file....that's a great way to lose a valuable program. There is no fix for this; just be careful in choosing your file names.

There is a further complication. Run this program using the same TESTFILE name as before.

```
10INPUT "File to read ";f$
20PRINT "^DOPEN ";f$
30PRINT "^DWRITE ";f$
40FOR x=1 TO 100
50PRINT "This is data line number ";x
60NEXT x
70PRINT "^DCLOSE ";f$
```

2 things will happen. The program will abort with an error message, and your file will still be open since the CLOSE command was not executed. The latest revision of SMARTBASIC allows you to partially recover from this problem by letting you issue the CLOSE command without the control-d. When your program aborts, before doing anything else, type the following:

```
CLOSE filename(the file name you used)
or
PRINT"^DCLOSE ";f$
```

There is disk activity and the file closes. Check the file size and you will see that it is 1K. If you take out your disk editor and consult the catalog, you will see that the bytes-used-in-last-block is close to if not exactly 1024. Run the program again using a file name that does not exist on the medium; this time everything goes normally provided you have sufficient room on the medium. You have now created a 3K data file.

Why did this happen? When the file was created, it only occupied 1K on the medium. The subsequent write operation tried to go over this 1K boundary and the file entry could not handle the extra size. This will happen even if the file was the last one on the medium. This is a drawback which could probably be fixed when the file is the last in entry in the directory, I just have not had the

ambition to try it. We can learn one lesson from this. When creating a file which will be updated later, you should blank fill it to a suitable length. Thus when you read in your data, manipulate it, and write it back, there will be sufficient room allocated in the directory.

Once a file has been created, how do you get at the information stored on the file? That's when the READ command comes into play. Recreate the data file which contained your name and some numbers, and run the following program:

```
10INPUT "File to read ";f$
20PRINT "^DOPEN ";f$
30PRINT "^DREAD ";f$
40INPUT name$
50INPUT df$
60INPUT number$
70PRINT name$
80PRINT df$
90PRINT number$
100INPUT "What do I do now ";a$
```

When you run the program, you will see a "?" appear on the screen for each INPUT statement. This is because the READ command has channelled normal keyboard input to the file and INPUT without a string prints a "?". Before we handle that problem, what happened at line 100? We got an OUT OF DATA error. That is because we did not close the file and BASIC is still trying to read input from the file. Since it has reached the end of the file, it aborts with an error....so let's close the file:

```
10INPUT "File to read ";f$
20PRINT "^DOPEN ";f$
30PRINT "^DREAD ";f$
40INPUT name$
50INPUT df$
60INPUT number$
65PRINT "^DCLOSE ";f$
70PRINT name$
80PRINT df$
90PRINT number$
100INPUT "What do I do now ";a$
```

Something is still going wrong! The CLOSE command appeared on the screen and I still get the annoying error message. Now we come back to those annoying question marks. Because they get printed on the screen, the CONTROL-D preceding the CLOSE command did not appear as the first character on the line. So let's get rid of the "?" by using a different INPUT statement.

```
10INPUT "File to read ";f$
20PRINT "^DOPEN ";f$
30PRINT "^DREAD ";f$
40INPUT "";name$
50INPUT "";df$
60INPUT "";number$
65PRINT:PRINT "^DCLOSE ";f$:REM make sure close is on a line
70PRINT name$
80PRINT df$
90PRINT number$
100INPUT "What do I do now ";a$
```

I have added a null string to each input statement which will cancel the "?". Also, the additional PRINT statement in line 65 makes sure that the ^D will appear at the beginning of a line.

Let's use the POSITION command to move to record number 3. POSITION is a way of advancing in a file by skipping the number of records specified. You can only skip forward. POSITION cannot be used to reset the file pointer backwards. Thus POSITION should have been named SKIP. The POSITION command clears any previous READ command and the READ command must be re-issued. The POSITION command includes the file name, a comma, and an "R" followed by the number of records to skip. Here's our new program.

```
10INPUT "File to read ";f$
20PRINT "^DOPEN ";f$
30PRINT "^DPOSITION ";f$",r2":REM skip 2 records
50PRINT "^DREAD ";f$
60INPUT "";number$
65PRINT:PRINT "^DCLOSE ";f$:REM make sure close is on a line
70PRINT name$
80PRINT df$
90PRINT number$
100INPUT "What do I do now ";a$
```

and our data file looked like this:

```
My name is Guy Cousineau
This data is being sent to testfile
1   .66666666  5.18  431460
```

Note that when the read program gets the third line, the numbers are read in as strings and it would be difficult to decode these numbers. One way is to use GET statements in your read routine. This modified program will read a number and print a new line after each one:

```
10INPUT "File to read ";f$
20PRINT "^DOPEN ";f$
30PRINT "^DREAD ";f$
40INPUT "";name$
50INPUT "";df$
60x= 1 REM read first number
70GET q$:REM read a character from the file
80IF q$>="0" and q$ <="9" GOTO 150
90IF q$="." GOTO 150:REM include decimal point as well
100IF nonum=0 THEN PRINT:New line if first non number
110IF nonum=0 then x=x+1:REM that's one more number
120nonum=1:REM make sure we don't do this again
130if x=4 GOTO 200:REM we have read the last one
140GOTO 70:REM read another character
150nonum=0:REM we have a number
160PRINT q$;:REM print out one digit
170GOTO 70
200PRINT:PRINT "^DCLOSE ";f$:REM make sure close is on a line
```

That was a lot of work! It would be even more work to interpret the digits and turn them into a numeric variable. The moral is: don't concatenate numbers on a line if you want to read them back as numbers! Print out each number on a separate line when you create your data file. The file will not be much bigger, just longer when you look at it.

Let's consider a practical example. The following program paints random blocks on a GR screen. The

latter half saves the screen to a file called GRSCREEN.

```
10GR
20FOR z=1 to 100
30COLOR=INT(RND(1)*16)
40x=INT(RND(1)*40)
50y=INT(RND(1)*40)
60PLOT x,y
70NEXT z
80PRINT "^DOPEN GRSCREEN"
90PRINT "^DWRITE GRSCREEN"
100FOR x=0 TO 39
110FOR y=0 TO 39
120PRINT SCRN(x,y)
130NEXT:NEXT
140PRINT:PRINT:"^DCLOSE GRSCREEN
```

The double loop from 100 to 140 interprets the screen colour and saves it to file. Later on, you want to restore that GR screen:

```
10GR
20PRINT"^DOPEN GRSCREEN"
30PRINT"^DREAD GRSCREEN"
40FOR x=0 TO 39
50FOR y=0 TO 39
60INPUT"";c
70COLOR=c
80PLOT x,y
90NEXT:NEXT
100PRINT:PRINT"^DCLOSE GRSCREEN"
```

When you write games, you may wish to have instructions that the player can view on screen prior to starting the game. If your program is very long, these PRINT or DATA lines will eat up valuable RAM space. Here's another application for a DATA file. Create an "A" file called GAMEDOC using the OPEN WRITE CLOSE commands. Just print a few blank lines into the file. Then, using SMARTWRITER, load in the GAMEDOC file and set your margins to 10 and 41. This will give you a 31 column screen compatible with SMARTBASIC. At the end of your file, put in an "@" character to signify the end of file. Do NOT use the "@" anywhere else in the file. Write in as many lines of text as you want, using any other punctuation. Then type in and RUN the following program:

```
10PRINT "^DOPEN GAMEDOC"
20PRINT "^DREAD GAMEDOC"
30GET q$
40PRINT q$;
50IF q$="@" GOTO 100
60IF q$="." THEN FOR x= 1 TO 1000:NEXT
70IF q$=CHR$(13) THEN FOR x=1 to 2000:NEXT
80IF q$="," THEN for x=1 to 500:NEXT
90GOTO 30
100PRINT:PRINT"^DCLOSE GAMEDOC"
```

This short routine will read in a document file of any length, and it takes up only a few lines in your program. Lines 50 to 80 put in pauses for periods, carriage returns, and commas. You may add in more pauses for other characters if you wish. You can also experiment with the wait values until you reach a comfortable reading speed.

As we have seen, the GET and INPUT commands are channelled to the file when READ is active. But what if I want some user input? Lets' say I want to give the opportunity to abort the file read without crashing the program. This can be done using the last key press register. First, make sure that there is a <CR> in the register prior to starting by adding these 2 lines:

```
5INPUT "Press <CR> to start, any other key will abort";x
85IF PEEK(64885)<>13 GOTO 100
```

Line 5 forces a <CR> by the use of an INPUT statement...this must be done prior to opening the file. Line 85 makes a jump to the CLOSE command if any key is pressed while viewing the documentation. You could even interpret a special character which would pause the listing, or even one to reduce or increase the pauses. We'll let you figure out how to do that.

Reading DATA files of unknown length can always cause problems. Where is the end of the file. Rather than rely on an end of file marker, you can use error trapping:

```
10PRINT "^DOPEN GAMEDOC"
20PRINT "^DREAD GAMEDOC"
250NERR GOTO 95
30GET q$
40PRINT q$;
60IF q$="." THEN FOR x= 1 TO 1000:NEXT
70IF q$=CHR$(13) THEN FOR x=1 to 2000:NEXT
80IF q$="," THEN for x=1 to 500:NEXT
90GOTO 30
95CLRERR
100PRINT:PRINT"^DCLOSE GAMEDOC"
```

The new line 25 sets an error trap to branch to the close routine. Note that pressing ^C will also generate an error that will close the file. The previous line 50 has been removed since we no longer need an end of file marker. Line 95 is crucial. If you do not clear the error trap, any subsequent errors would jump back to the close command and could lock up your program. You can use multiple error traps to make sure your program does not crash at any stage of a data file access operation:

```
5ONERR GOTO 200
10PRINT "^DOPEN GAMEDOC"
20PRINT "^DREAD GAMEDOC"
250NERR GOTO 95
30GET q$
40PRINT q$;
60IF q$="." THEN FOR x= 1 TO 1000:NEXT
70IF q$=CHR$(13) THEN FOR x=1 to 2000:NEXT
80IF q$="," THEN for x=1 to 500:NEXT
90GOTO 30
95CLRERR
100PRINT:PRINT"^DCLOSE GAMEDOC"
110END
200PRINT "Could not open data file"
210PRINT "Press 'q' to abort"
220PRINT "or any key to retry"
230GET q$
240GOTO 5
```

In a situation where you have accumulated a lot of data in memory and you are trying to write it out to a data file, these kind of traps are essential.

Back to an earlier problem. Remember in the last article when we tried to create our DATA file twice? Each OPEN WRITE sequence resets to the start of the file. But what if you want to add data to a file? Here's where you use APPEND. APPEND is smart: it knows you want to WRITE to the file name supplied. Thus OPEN and WRITE are replaced by APPEND. APPEND also knows that your file might wind up being bigger: it therefore recopies the entire file to the end of the directory prior to the write operation. This may take several seconds for a large file, (several minutes on tape). Try out the following program:

```
10INPUT "File to append ";f$
20PRINT "^DAPPEND ";f$
30INPUT "data (<CR> to end ";x$
40PRINT x$
50IF x$<>"" GOTO 30
60PRINT:PRINT"^DCLOSE ";f$
```

Run this program several times, adding a few new lines of text". Once that is completed, take out your favourite directory management utility (mine is FILEMANAGER) and look at the tail end of the directory. You will see several versions of your data file, a few bytes bigger each time. You can clearly see that several append operations will quickly gobble up all the remaining directory entries on your medium.

One additional warning about writing to data files. When SMARTBASIC opens a file at the end of the directory, it reserves all the remaining space to the end of the disk/tape for the file. If you fail to close the file properly, the storage medium will be rendered unusable.

MON and NOMON refer to the program monitor. There are 4 control functions:

```
MON Cmonitor commands
MON Imonitor INPUT
MON Omonitor OUTPUT
MON Lmonitor LISTING
```

Several MON or NOMON instructions can be issued at once with something like:

```
NOMON c,o
```

to turn off the monitor for output and commands. Command monitor will echo the OPEN READ WRITE APPEND commands to the screen; this might be useful in debugging. The INPUT monitor will echo characters read in using INPUT or GET statements. The OUTPUT monitor will echo the data written with PRINT statements. The LIST monitor will echo a program as it is being loaded. Try MON L and follow with LOAD programname.

TECHNICAL SECTION:

OPEN executes at 24497 (5FB1). It gets the file name and record size if any (see next article on random files). It tries to open the file and if there is none a new file is created. It then allocates either of the 2 file buffers to the opened file.

CLOSE executes at 24612 (6024). It gets the file name and the drive, and checks to see if the file is using buffer 1 or 2. It then trims the file and flushes out any remaining characters. It then restores normal keyboard input and closes the $$$$1 (or $$$$2) temporary file name. If the close is successful, the file is renamed to the requested name and releases the file buffer.

READ executes at 22049 (5621). It checks the file name against the buffers allocated. It then decides if the file is random or sequential and jumps to the appropriate handling routine. For

sequential files, it is just a matter of checking the buffers and setting the READ-FROM-MEDIUM vector which channels INPUT and GET to read from the file.. For random files, a complex series of calculations are made to determine the location of the requested record.

WRITE executes at 22455 (57B7). It also checks the file name buffer and decides which output routine to select.

APPEND executes at 21477 (53E5). After the syntax checking, it tediously copied the file byte by byte to the temporary file at the end of the directory. It then jumps to the middle of the WRITE routine to set up the buffers.

POSITION executes at 21715 (54D3). It check that the file is opened and looks for the comma and "R" syntax characters. It then sets a counter for the number of records to skip. The file is read and carriage returns counted until they reach the requested skip.

MON and NOMON share the same routine. They have different entry points which set up pointers to the execution vectors for the CIOL options. NOMON starts at 23042 (5A02) and MON at 23047 (5A07). It looks for the CIOL option character and aborts on error. Once a legal option is found, the execution address is extracted and executed. The routine takes over again to look for another option character. This continues until the end of the command or an error is detected.

RANDOM ACCESS FILES

Random access files are supposed to be the best way to access data in various parts of a data file. Unfortunately, the COLECO SMARTBASIC has serious problems creating random files. Reading them, however, works as expected and reasonably fast.

Random files are so called because you can skip anywhere in the file forward or backward. In order for this to work, however, each record (entry) in the file must be a fixed length. That is the only way SMARTBASIC can figure out where to go when you ask for a specific record. Random files are opened and read in a very similar fashion to sequential access files; you may wish to refer to that article for additional information. The differences will be explained in this article.

Whether READING or WRITING, a random access file must be opened by specifying the record length. Thus when you create a random access file, make sure you remember the specified record length. If you use an incorrect record length, you will have difficulty reading the file accurately. Note also that record length cannot be modified once the file is created; be sure to plan carefully. The OPEN SYNTAX is:

PRINT "^DOPEN file,Lx"

where file is a legal file name and x is the record length. Thus if my file is called DATA and has a record length of 20, I would enter:

PRINT "^DOPEN data,L20"

Now my file is opened and I can red or write. Note also the "^D" which is the imbedded CONTROL-D character we used in sequential file access.

When writing to a random file, you use the WRITE command but also specify the record number to write to:

PRINT "^DWRITE file,Rx"

where x is the record number. This would be fine if we had a print statement to refer to each record in the file but is would surely be very tedious. Thus we need to express the record number using a variable. Look at the following program which writes out 10 records based on user input from the keyboard:

```
10REM random write demo
20PRINT "^DOPEN datafile,L255":REM 255 is max length
30FOR x=0 TO 9:  REM repeat for 10 records
40PRINT "Record # ";x;:INPUT ":  ";x$(x) :  REM get the data
50NEXT x
60FOR x=0 to 9:  REM write out the 10
70PRINT "^DWRITE datafile,R";x :  REM position file
80PRINT x$(x) :  REM send data to file
90NEXT x
100PRINT "^DCLOSE datafile"
```

Just as you can print a combination of literal strings and variables (line 40), you can use a variable to specify the record number to WRITE to (line50). Remember to enclose the <comma><R> within the quotes without a space. Note however, that we had to read in all the data prior to giving the WRITE command. This is because the WRITE command re-channels the PRINT command to the file; this makes it impossible to print anything to the screen while file write is activated.

As I discussed earlier with the APPEND command, the random write command will create a new file for each write operation. Thus the program above will create 10 files (8 of which are deleted) each of increasing length. The WRITE command should be smart enough to recognize a write within the existing file (i.e. modifying a record). If so, the multiple file problem could be reduced by writing the LAST record FIRST and follow by modifying the other records. I have found no suitable way of effecting this modification to SMARTBASIC.

The RANDOM READ command uses a syntax similar to write to randomly access any record in the file. Our first example reads the 10 records we just wrote in reverse sequence and echoes them to screen:

```
10REM random read demo
20PRINT "^DOPEN datafile,L255" :  REM same length as created
30FOR x=9 to 0 STEP -1
40PRINT "^DREAD datafile,R";x :  REM position file
50INPUT "";x$
60PRINT "record ";x;" ";x$
70NEXT x
80PRINT "^DCLOSE datafile"
```

In order for READ to be effective in accessing a data file interactively, we must find a way to get user input. If you recall the discussions on READ, it assigns the INPUT and GET commands to the file rather than the keyboard. We will therefore use the LAST KEY PRESS REGISTER to manipulate user commands. Let's build on our existing program.

```
10REM user controlled random read
15POKE 16149,255 :  POKE 16150,255 :  REM clear POKE limit
20PRINT "^DOPEN datafile,L255" :  REM same length as created
30PRINT "record number or 'q' to quit"
32p=PEEK(64885):  IF p=ASC("q") GOTO 80
34x=p-ASC("0") :  REM turn key press to number
```

```
36IF x<0 OR x>9 GOTO 32
38POKE 64885,0 :  REM clear current command
40PRINT "^DREAD datafile,R";x :  REM position file
50INPUT "";x$
60PRINT "record ";x;" ";x$
70GOTO 30
80PRINT "^DCLOSE datafile"
```

Note that without line 38, the program would loop, continuously reading the selected record until another key is pressed. The program area between lines 30 and 40 could be expanded to interpret a 2-digit record number and/or decode other instructions or commands. The expansion is left up to you.

ROUTINE ADDRESSES

OPEN starts its execution at 24997(5FB1). It starts by getting a file name and then calls a routine 20730(50FA). This is a double duty routine which looks for a comma followed by either an A or an L. If this combination is found, it checks the next character for an $ which is the HEX prefix. Yes you can specify the record length in HEX if you wish. Once the number is decoded, it is placed either in FILE LENGTH (16793-4199) or FILE ADDRESS (16791-4197) if an L or A was typed. If no file length is specified, the FILE LENGTH data byte is set to 0. This byte will be checked by the read command.

READ starts at 22049(5621). It starts by getting a file name and checks for another character on the command line. If there is none, sequential read is assumed and the routine exits by setting the appropriate file buffers and input routines. If a character is found then a CALL to 22923(598B) is made. This routine looks for the COMMA, and expects the next character to be a B or an R (probably another Apple compatible syntax). If both are not found an error is generated. The next step is to get the record number which could be in HEX or DECIMAL. Once the record number is decoded, a complex routine at 22181(56A5) is executed. It sets up a double loop to read from the start of the file for RECORD_NUMBER*RECORD_LENGTH to correctly position the file. This routine might be made more effective is instead it calculated the BLOCK NUMBER of the file based on the parameters and skipped directly to it. Still, it is reasonably fast...from disk.

WRITE starts at 22455(57B7). It starts by getting a file name and checks for another character on the command line. As with read, if one is found a call to 22923(598B) is made to check syntax and get the record number. If the parameters are OK then a jump is made to the middle of the APPEND routine at 22562(5822). This is the part that makes a new file every time random write is invoked. All characters up to the selected record number are copied to a temporary file (at then end of the directory). This differs from the regular APPEND which copies the entire file. Once the partial file is copied, the WRITE_TO_FILE vector is set for subsequent PRINT operations.

CLOSE starts at 24612(6024). If the file was a random file, characters from the current record number (in the original file) to the end of file are copied to the new version. Then all file buffers are closed and the temporary file names renamed.

When file write is active, PRINT is channelled to 21839(554F). This simply extracts the MON O (21869-556D) or NOMON O (21874-5572) vector and jumps to the proper routine. This routine is quite complex but does not seem to check that the requested write is not longer than the file's specified length. Thus discipline is required when writing to random files that each RECORD is trimmed to the correct length prior to writing it to file.

When we discussed the PR command we referred to channelling INPUT and OUTPUT vectors. What this means is that BASIC has the ability to channel INPUT, GET, and PRINT to a file, printer, screen, or other

device (if such a device is defined by the programmer). BASIC stores a series of vectors which point to the PRINTER, SCREEN, and FILE routines and places them accordingly in the ACTIVE vector which is located at 16201-16202. The printer/screen vector is stored at 16199-16200 when a file has been opened for write; that value is restored once the file is closed. This gives us the opportunity to circumvent the PRINT TO SCREEN problem when writing data. Consider this revised WRITE routine:

```
10REM interactive random write demo
20PRINT "^DOPEN datafile,L255":REM 255 is max length
30FOR x=0 TO 9:  REM repeat for 10 records
40PRINT "Record # ";x;:INPUT ":  ";x$ :  REM get the data
70PRINT "^DWRITE datafile,R";x :  REM position file
80PRINT x$ :  REM send data to file
85POKE 16201,PEEK(16199):POKE 16202,PEEK(16200)
90NEXT x
100PRINT "^DCLOSE datafile"
```

When file read is in progress, BASIC redirects INPUT and GET to read from the file. This is easily accomplished since the GET_A_CHARACTER routine at 12137 (2F69) checks the current IN vector to use to fetch a character for GET or INPUT. When FILE READ is active, this routine is located at 21843(5553). It extracts the MON/NOMON I vector and makes the appropriate jump. MON I just CALLS NOMON I and then prints the character; it is at 21847(5557). NOMON I starts at 21981(555D).

The same kind of approach can be used with READ, but 2 patches must be made. The READ command rechannels the IN vector at 16197 and the PRINT_TO_SCREEN vector at 16795. The following program readjusts both those vectors to permit input from the console for commands.

```
10REM user controlled random read
20PRINT "^DOPEN datafile,L255" :  REM same length as created
30INPUT "record number or 'q' to quit";r$
32if r$="q" GOTO 80
34x=VAL(r$):IF x=0 and r$<>"0" GOTO 30
36IF x<0 OR x>9 GOTO 30
40PRINT "^DREAD datafile,R";x :  REM position file
50INPUT "";x$
55POKE 16197,PEEK(16195):POKE 16198,PEEK(16196):REM IN
56POKE 16203,PEEK(16795):POKE 16204,PEEK(16796):REM SCREEN
60PRINT "record ";x;" ";x$
70GOTO 30
80PRINT "^DCLOSE datafile"
```

Note in both these cases that the vectors are reset AFTER the READ/WRITE command. This ensures that the currently active vectors have been saved in the expected memory locations.

BINARY FILES

Prior to explaining the so-called binary file commands, a bit of explanation about file structure. The EOS uses 2 standard file types. The "A" file types signify ASCII files which can be read and manipulated by a Word Processing program. The "H" files have a HEADER (thus the H) which clarify the

structure of the file. All H files start with a 3-byte pre-header. There are 3 KNOWN types of "H" files:

```
bytes0 1 2 3 4 5 6 7 8
SMARTWRITER00 01 01 ................
BINARY01 00 02 lo hi ..........
FlashCard??  ??  16 ................
```

SMARTWRITER files have a 256 byte header which is indicated by the first 2 bytes of pre-header. The third byte of pre-header (1) identifies the type of file (SW). Starting at byte 3, we have page format, spacing, margins, tab arrays, etc. Note that the actual file starts not at byte 256 but 259 (remember the 3 bytes of pre-header.)

BINARY files show a header length of 1 and use a file type of 2 in position 2. Bytes 3 and 4 are the lo-hi address to load in the file. The data starts in byte 5 of the file. Binary does not necessarily mean un-readable data, it just means that the data is literal and is loaded exactly as it was written to the storage medium.

The commands BSAVE BLOAD and BRUN are used to access binary files. These commands can be given directly from the console or within programs using the CONTROL-D buffer. Refer to the article on DATA FILES for the elementary syntax.

BSAVE is used to save a binary file. It captures the contents of memory and saves it to a file. The syntax is:

```
BSAVE filename,Axxxx,Lyyyy,Dn
```

where filename is any legal file name, xxxx is the memory address where the image starts at, yyyy is the length of the image, and n is the optional drive number. It can be used to save program data, machine language programs, or SMARTBASIC itself.

Why use binary files? Firstly, binary files are not interpreted when they are loaded in; this makes them load much faster. Secondly, they will be much shorter than their ASCII equivalents. Let's look at one short example using a machine language program:

```
10FOR x=0 to 10:  READ y:  POKE 30000+x,y:  NEXT
20DATA 100,2,20,11,19,201,195,31,231,149,201
```

Even when program lines are concatenated, this routine requires 95 bytes. If the DATA had been saved as a DATA file as outlined in the article on DATA files, it would look like:

```
100
2
20
11
19
201
195
31
231
149
201
```

But it would still occupy 39 bytes and require several hundred bytes of program to read the data in. Saving it as a binary file will require 11 bytes plus 3 bytes of pre-header and 2 bytes for the load address: a grand total of 16 bytes. While it may appear insignificant for such a short file, you can

appreciate the enormous savings when the data in question is several hundred bytes long.

BLOAD will reload a previously saved image. While it uses syntax similar to BSAVE, it is not necessary to specify the load address. Since the load address was saved as part of the file, SMARTBASIC will know where to place it:

```
BLOAD filename
BLOAD filename,Axxxx,Dn
```

You can override the default load address and RELOCATE your DATA or ROUTINE by specifying the load address. If the file to load resides on another medium, you should be able to load it by specifying the drive only on the command:

```
BLOAD file,D2
```

Unfortunately, SMARTBASIC aborts if the next thing after the file name is not a load address. See the notes at the end for an easy fix.

BLOAD will only load a file with the proper header (01 00 02) as illustrated above. This helps prevent accidents like trying to load a Word Processing file which could damage the operating system or crash BASIC. If the file type is improper you will get a "File Type Mismatch" error.

BRUN will LOAD and RUN a binary file. It must be used exclusively to execute machine language routines. It uses the same syntax as BLOAD.

This all sounds quite simple but it becomes complex when we try to use it correctly. Firstly we have a problem with the CONTROL-D buffer. This buffer, located 17016(4278) is used to store the characters printed after a CONTROL-D prior to their execution. The buffer, however is only 23 bytes long. This is fine for commands such as:

```
BSAVE abc,a100,l55
BLOAD sprites,a30000,d2
BRUN sound,d5
```

A long (and perfectly legal) command like:

```
BSAVE adventure,a12345,l10000,d1
```

is 33 characters long (including the line terminator. If you try and use such a command from a program you will get a "Control Buffer Overflow" error message. Relocating the buffer can be quite complex since there are several references to the start and end of buffer. I have found one trick which involves moving the start of the buffer to a lower value. While the position appears to be reasonably arbitrary, we must make sure that at least 33 bytes are available from the start of the buffer. I have come upon this fix which moves the buffer to 16156(4084):

```
POKE 17040,132: POKE 17041,64
```

Now on to a few practical examples. Let's take saving a GR screen which was covered in the DATA article with this program:

```
80PRINT "^DOPEN GRSCREEN"
90PRINT "^DWRITE GRSCREEN"
100FOR x=0 TO 39
110FOR y=0 TO 39
120PRINT SCRN(x,y)
130NEXT:NEXT
```

```
140PRINT:PRINT:"^DCLOSE GRSCREEN
```

This will create a data file with 1600 2 or 3 byte entries: firstly a number from 0 to 15 and secondly a carriage return. Thus our image file will be a touch over 3K and occupy 4K on disk/tape. Now consider this routine:

```
100LOMEM: 29600
110FOR x=0 to 39
120FOR y=0 to 39
130POKE 29600+40*x+y,SCRN(x,y): REM save in memory
140NEXT: NEXT
150PRINT"^DBSAVE GRSCREEN,a29600,l1600"
```

To reload the screen, we would do the following:

```
200LOMEM: 29600
210PRINT "^DBLOAD GRSCREEN"
220FOR x=0 to 39
230FOR y=0 to 39
240COLOR= PEEK(29600+40*x+y)
250PLOT x,y
260NEXT: NEXT
```

This approach uses a data file which is 1605 bytes long (remember the 5 bytes of header).

That example forced you to POKE data which is not normally stored in memory. This may not make a binary file the best option. If, however, it was necessary to refresh the screen based on the same data at several occasions within the program, there might be an advantage to POKEing the data in memory.

Another application is SPRITE tables. A set of 32 sprites requires a 128-byte table for the coordinates and position. They also require 256-byte table for the sprite shape definitions. It will take several DATA statements and a lengthy series of POKEs to install all this data in memory. Once it is loaded, the DATA is no loner required, it will still occupy about 2K of valuable memory. The preferred approach is to create a LOADER program which will do only that: POKE in the sprite tables (say at 29000). Then the data is saved with:

```
BSAVE sprites,a29000,l384
```

Which occupies only 389 bytes of disk space and can be loaded quickly with:

```
BLOAD sprites
```

Since sprites can be loaded at any memory address (provided your sprite routine knows where they are), you can quickly relocate them with:

```
BLOAD sprites,a30000
```

I mentioned earlier that you could use binary files to save SMARTBASIC. If you have made several modifications to your system with a lengthy HELLO file, you may find it takes several seconds to install your modifications. Once this is done, simply type:

```
BSAVE basic,a100,l44444
```

Even though BASIC is only 32K long on the tape/disk, you must now save a longer segment to capture the variable commands and other program pointers. Although I could probably calculate the exact value, 44444 is easy to remember. To reload you modified BASIC, put the following in your new HELLO file:

```
10PRINT "^DBRUN basic"
```

The reason we use a BRUN is to be sure BASIC resets itself. Don't forget to save your old HELLO file; it will come in handy if you make additional changes to your SMARTBASIC.

ROUTINE ADDRESSES

Note that Input/Output commands do not have separate PARSE and EXECUTION vectors. All processing is handled in one pass.

BSAVE like other media commands executes from a roundabout way either from an immediate mode vector at 20022 (4E36) or from 19733 (4D15). The actual routine is at 20849 (5171). It calls separate routines to get the name, address, length, and drive. It then assigns a temporary name ($$$$1 or $$$$2) and figures out the size of the file by adding 5 to the prescribed length. It then sets up the header and writes the save/load address to the file. The next step is to write the data and finally close the file.

BLOAD gets its execution address from 20029 (4E3D) or 19740 (4D1C) and executes at 20993 (5201). It starts by getting the name and checks for another character. If there is none, it sets a flag to accept the default load address. If the character after the expected comma is not an 'A', an error is issued without checking if the character was a 'D' for drive specification. To correct this, POKE 21019,11. The next step is to get the source drive name (if any) and check that the requested file is indeed an 'H' file. The load address is read from the file but replaced with the user's address if it was supplied. If the file is loaded without error, HL is loaded with the load address and a return is made to the caller.

BRUN gets its execution address from 20036 (4E44) or 19747 (4D23) and executes at 21140 (5294). It simply calls BLOAD and follows with a JUMP to (HL). You may have encountered some programs that include machine language routines that may be BRUN. Some of these may have warnings to BLOAD and then CALL the routine. Since BRUN does not save any registers, the routine exits may be unpredictable in some instances. Those are usually the fault of the programmer who wrote the routine and not of BASIC itself. Nevertheless, heed their warnings and use BLOAD / CALL for your own peace of mind.

INTEGER VARIABLES

Have you ever seen a program with lines like c%=a%+2*e%? The PERCENT sign tells SMARTBASIC that your numbers are signed integers in the range of -(2^15) to 2^15. So how much is that? from -65536 to +65535. You may wonder where the extra number went; don't forget we need to express zero as well. Since 0 is technically a positive number, we have 65536 positive numbers and the same number of negatives.

So why should I use integer variables anyway? One important reason: space!

When integer variables are stored in memory, each takes 2 bytes compared to a floating point number which takes 5 bytes. Consider the amount of memory required by the following arrays which might be used to represent the 4 suits of 13 playing cards in a deck:

```
ARRAY RAM USEDDIFFERENCE
DIM a(4,13)  355
```

DIM a%(4,13) 145 40%

Not all numbers can be defined as INTEGER variables. DEF, and FOR require a real variable (floating point) because of the nature of their execution. You can, however change an integer variable into a real one with:

a=a%(both a and a% are different variables)

One might expect that operations on integer variables are faster, but my testing reveals that they take about 10% longer which seems rather odd. This is probably due to the fact that integer operations cannot take advantage of exponents to speed up calculations. That 10%, however, seems trivial when integer arrays take less than 1/2 the space of floating point arrays.

Integer variables can also be used as shortcuts for calculations. Consider the determination of low/high bytes for a memory address:

high=int(value/256)high%=value/256
low=value-256*highlow%=value-256*high%

The first equation on the right is simpler and guarantees an integer result. Remember the round off error problems which occur particularly when you divide a number by 100? Try the following program:

```
100FOR x=1 TO 100
110a=x/100
120a%=x/100
130b%=x-a%*100
140PRINT a,a%;".";b%
150NEXT
```

You will notice two round off errors on the left side, the right side always has the correct answer. If you are writing a program to print out dollars and cents, you can use this technique for prettying up your display and ensuring you always have exactly two digits to the right of the decimal. Let's presume you have done all your calculations in floating point and now have a variable called value that you wish to print:

```
200a%=valuetake the integer
210b%=100*(value-a%)bring the pennies in
220PRINT "$";a%;".";b%;note the semicolon at the end
230IF b%=0 THEN PRINT "0";add the double zero
240PRINTmove to new line
```

SHAPE TABLES

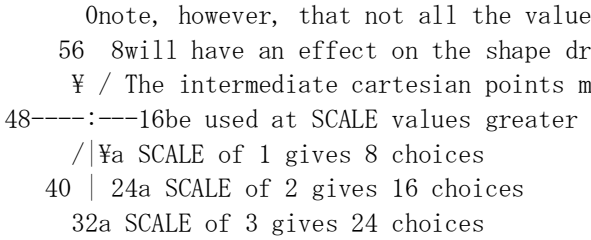
After taking a short break from BASIC commands, we must now return to take care of our last set of commands to cover; they deal with to shape tables. Although the related commands were briefly covered in the chapter on HI RESOLUTION GRAPHICS, we will go over them again in a bit more detail. Remember that shape tables can only be used in HGR and HGR2 mode. Consequently, you must be in one of these modes in order to use the commands listed below.

SCALE decides which multiplication factor to apply to each shape instruction in the shape table. Thus a scale of 1 is normal size, 2 is double size, 10 is ten times the size, etc. SMARTBASIC's default SCALE value is 255. Have you ever designed or used shapes in a program and forgotten the SCALE statement? The shape splatters all over the screen and it seems you have to wait forever before you get an opportunity to break the program. Do NOT use a scale value of 0; it is actually interpreted as

255. So remember to use a SCALE command at the beginning of your programs that use shape tables.

Some shapes scale up very well and others do not. A shape consisting of straight lines and square corners will scale relatively well provided attention was paid to DRAWing it correctly. More on that in the next article.

ROT specifies the rotation of a shape; it can have values from 0 to 63 according to the diagram below:



Thus you can use the ROT command to rotate your shape on the screen. Remember, however, that ROT does not achieve a true rotation about the centre of a shape. The start coordinate (i.e. DRAW AT x,y) is always the same and the ROT command decides in which direction to move. To illustrate this, use the default shape table that comes with BASIC and run the following program. If you have defined your own shape table, deinstall it.

```
10HGR:HCOLOR=7
20FOR s=1 to 8:SCALE=s
30FOR r=0 to 63:ROT=r
40DRAW 1 at 128,80
50print "scale ";s,"rotation ";r
60XDRAW 1 at 128,80
70NEXT:NEXT:END
```

Firstly, you will notice that the higher the SCALE value, the more individual rotations can be represented. You should also notice that at higher SCALE values, the rotations are not the same size but actually a bit larger than the original. As far as the rotation is concerned, the shape seems to spin about its centre. Now change the shape number from 1 to 2 in lines 40 and 60 and run the program again. This time you don't get a pretty shape; neither does it rotate about its axis. You will probably notice the same erratic rotation on your own shapes unless you have drawn them from the centre out as I will explain in the next article.

DRAW will paint a shape in using the current HCOLOR value, e.g. DRAW 1 at 100,100. The supplied coordinates indicate at which point to start drawing the shape. The DRAW command is smarter that most other graphics commands in that it can wrap itself around the screen. This is particularly evident at larger scale values; as you push a shape off the right side of the screen, its leftovers appear on the left side. There is also an undocumented syntax for the DRAW command: DRAW s where s is the shape number. When you use this command the shape is drawn starting where the previous shape drawn left the "cursor" or "pen"...

```
10HGR:HCOLOR=7:SCALE=1:ROT=0
20DRAW 1 at 100,100
30FOR x= 1 to 10
40DRAW 1 :  REM draw next where I left off on the first
50NEXT
```

This drawing technique has some possibilities and you can use it to your advantage if you design your

shape tables with that in mind.

Remember that the DRAW command is not very smart in one respect: it does not check if the shape is within the range of the shape definition table. In our first example, we used shape number 2 to illustrate the off-centred rotation. But there is no shape number two in the default shape table. Calling for a shape which is outside the shape table boundary will usually result in something strange. It will extract bits from memory in and execute them as if they were shape definitions. This will continue until a zero is reached which may take some time. Particularly when you use a variable to decide the shape to be drawn, be sure your value is always within range.

XDRAW will erase a shape at the supplied coordinates. It uses the same syntax as the DRAW command and a neat trick which ERASES rather than paints another colour. The HCOLOR is set to 128+current_colour, the DRAW command is called, and the HCOLOR is reset to its original value. When the VDP processing sequences take over, they interpret a COLOR above 127 as TURN_THE_PIXEL_OFF. You can use this technique to erase using the HPLOT command:

```
100POKE 16777,PEEK(16777)+128:REM set to erase
110HPLOT a,b to c,d
120POKE 16777,PEEK(16777)-128:REM restore color
```

SCALE executes at 2CD1 (11473). It fetches the next 'number from the token line and calls a routine at 66E8 (26344) to do the rest of the work. This is simply to check if the value is non-zero. A non-zero value is immediately placed in 417D (16765). If the value is zero, which includes illegal input, the SCALE is set to its maximum of 255.

ROT executes at 2CC3 (11459). It fetches the next 'number from the token line and calls a routine at 66DD (26333) to do the rest of the work. First the ROT value is ANDed with 63 which makes a ROT of 64 the same as a ROT of 0 etc. The resulting value is divided by 8 to determine the quadrant of rotation. A series of jumps is made based on the quadrant and the shape definition modified according to the degree of rotation.

DRAW begins its execution at 2C5E (11358). It gets the shape number and check to see if coordinates have been supplied. These coordinates are checked for the x maximum of 255 (redundant) and the y maximum of 191. This check does not take into consideration that you may be in GR mode where your maximum is 160. The DRAW with no coordinates continues at 66CD (26317) and without at 67DC (26588). The latter routine extracts the LAST USED XY from 417B (16763) and jumps into the middle of the DRAW routine. This one begins by moving the incoming coordinates from BC to DE and proceeds to draw SHAPE C at DE.

The first thing to do is find the VRAM position corresponding to the coordinates and copy into memory the 'pattern' already stored there. It will be modified as the shape is drawn and when the DRAW moves across a pattern boundary, the pattern is re-written and the next one fetched.

With the proper pattern and corresponding color table in memory, the next step is to fetch the address of the desired shape from the shape index table (we will discuss that in the next article). Each bit of the shape table is read in, interpreted, decoded, rotated, applied to the VRAM pattern in memory, verified for color, etc. It is rather complicated to figure out exactly who does what but the code stretches from 6825 to 6903 (26661-26883).

XDRAW begins at 2C94 (11412). It uses the same approach as above except that it jumps to secondary routines 66B9 (26296) with no coordinates and 6904 (26884) with coordinates. These two routines check for HGR mode, set the high bit of HCOLOR -> 4189 (16777), call the corresponding DRAW routine and the reset the HCOLOR bit.

SHAPE TABLE STRUCTURE

But what is a shape table? It is a structured array of bits which tell SMARTBASIC how to draw a shape. The basic structure in memory is: number of shapes1 byte
unused1 byte
offset to shapes2 bytes * number of shapes
shape definitionsunlimited
The number of shapes can vary from 1 to 255.

The second byte is unused and its value is unimportant. You can use it for whatever purpose you wish.

Starting at byte 3 are a pair of bytes for each shape in the table. They will give the offset of the shape. This offset is calculated from the start of the shape table (number of shapes) and not from the end of the jump table. That is the address of the shape table located at 417E-417F (16766-16767).

The next set of bytes are the shape definitions. Each bit means something specific until a null-byte is encountered - the end of the shape definition.

Before we discuss shape definitions, you may wish to run and EXAMINE this program to understand exactly how the components of a shape table interact:

```
 10LOMEM:40000
20INPUT"shape table to analyze ";f$
30PRINT chr$(4);"bload ";f$;",a30000"
35REM POKE in the address in memory
40POKE 16767,30000/256:POKE 16766,30000-256*PEEK(16767)
50HGR:SCALE=1:ROT=0:HCOLOR=7
60ad=PEEK(16766)+PEEK(16767)*256
70PRINT "Shape table is located at ";ad
80ns=PEEK(ad):PRINT"It has ";ns;" shapes"
90PRINT "press a key to continue";:get q$
100for x=1 to ns
110so=PEEK(ad+2*x)+PEEK(ad+2*x+1)*256:REM shape offset
120sa=ad+so:rem shape definition starts here
130PRINT:PRINT "Shape ";x;" Definition is ";
140p=PEEK(sa):if p<>0 THEN PRINT p;" ";:sa=sa+1:GOTO 140
150DRAW x at 100,100:GET q$:REM wait for a key
160XDRAW x at 100,100
170NEXT x
```

Now that we know the structure of the table, let's look at the bytes required to define a shape. Shapes are a combination of plot and move commands. The PLOT instruction can be compared to LOGO's 'pen up' 'pen down' command; it controls whether something will be drawn or not. The MOVE instruction decides if the pen will move up right down or left. Since the first is a true/false condition, it only requires 1 bit. The second is as follows: up00right01
down10left11

The PLOT/NOPLOT instruction precedes the move instruction, creating sets of 3 bits. To draw the letter "L" at a small scale I would need the following:

plot-go-down plot-go-down plot-go-down plot-go-right

110 110 110 101

This presents a problem when you want to put these together in groups of 8 bits to form a byte -- you can only put 2 and 2/3 instructions into one byte. For decoding purposes, each byte is presumed to be 9 bits long and the top bit is presumed to be a zero (or a don't plot instruction):

```
(8) 7 6 5 4 3 2 1 0
  P M M P M M P M M
 ¥  /  ¥  /  ¥  /
 three  two   one
```

Instruction number one is placed in the lower three bits, the second in bits 345, and the third can be placed in the top two bits if it is a NOPLOT instruction. If the third is a PLOT, just place a 00 in bits 6 and 7 which does not mean GO UP, it means IGNORE. In the third position then, you can only give NOPLOT instructions which move right, down, or left. The "L" shape illustrated above would be represented in the following fashion:

```
001101100010111000000000
 ¥ /¥ /  ¥ /¥ /
  2  1    4  3
```

Each instruction is read in groups of three starting at the right of each byte. The last ZERO byte ends the shape definition. The decimal values for these instructions are 54,46,0. The terminating zero is an essential part of the shape definition. Without it BASIC would keep on drawing forever. Be careful not to put two NOPLOT-GO-UP instructions in the same byte since their code is zero, it would be interpreted as the end of shape.

How do you put this all together into a shape table? Let's assume I want to create a shape table with two shapes, the "L" being shape number one:

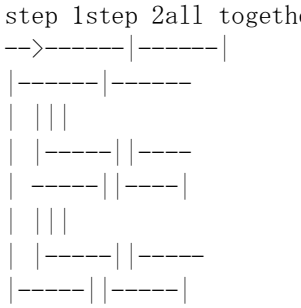
```
BYTEVALUEMEANING
002two shapes in table
100unused
206¥___shape one starts at byte 6 from start
300/
409¥____shape two starts at byte 9 from start
500/
654¥
746 >---data for shape one
800/
9xx----shape two starts here
```

When drawing complex shapes, you can plan your strategy to make the most effective use of the data bytes. Let's look at a large "E" as an example:

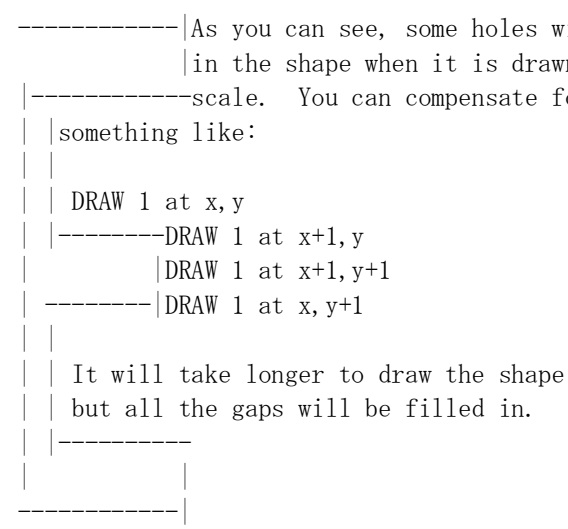
```
XXXXXXX
XXXXXXX
XX
XXXXXX
XXXXXX
XX
XXXXXXX
XXXXXXX
```

The NOPLOT instructions are only required when you need to skip over some WHITE SPACE (or you painted

yourself in a corner). If I start in the top left and go down the left side, right one and back up again, I could be in trouble. After zig-zagging through the top part of the E I will have to jump over some white space to do the middle leg and again for the bottom. A bit of analysis results in the following pattern which wastes no instructions:



When you look at it on paper, it might look a bit funny but at a scale of 1 it will look perfect. As you increase the scale, however, it will begin to look lop sided and may have some holes in it. Following is a representation of the "E" as drawn above at a scale of 2:



Another way to overcome the problem is to pick a drawing pattern which will minimize the HOLES when the shape is drawn at a higher scale. You may even wish to retrace some corners in two different directions. The shape definition will be larger but it should be prettier regardless of the SCALE used. If your shapes will only be drawn at a SCALE of 1, you need not worry about this problem and just pick the most economical path.

In the previous article, I mentioned the off-centre rotation of shapes. To make a shape rotate evenly about it's centre, you must start defining your shape in the centre of the grid. You may wind up with a longer shape definition, but it will rotate neatly.

I also previously mentioned that DRAW can be used with no x,y coordinates. In that case, the DRAW commences where the last shape ended. If you want to make some letters to use on an HGR screen, you can start defining all your shapes in the upper left and terminate in the upper right, even if this means using NOPLOT instructions at the end (watch out for a double NOPLOT-GO-UP). Once you have positioned your first letter, the rest can be DRAWN with no coordinates since the PEN will be properly placed. You just need to keep track of the approximate position to make sure it does not wrap around the right edge of the screen.

MAKING SOUND

As you have probably heard in games, ADAM is capable of creating complex sounds. This article will explain how to make those sounds from any program which lets you write and use a machine language routine. Although the explanations given here apply to making sound in BASIC, they can also be adapted to any other environment, including CP/M or TDOS.

All sound generation is made by sending values to the sound port (255). Those values are interpreted and executed by the sound processor chip. The first thing we need is a machine language routine to talk to the sound chip. It is quite simple:

```
3E 00LDA,0;put value in A
D3 FFOUT(255),A;send it to sound port
C9RET;go back to BASIC
```

The HEX values corresponding to the instructions are listed on the left. In decimal they are: 62,0,211,255,201. Now we need a program to POKE these values in memory. I will use 28000 as the base address, but you can use any available memory area:

```
10LOMEM:28005
20FOR x=0 to 4
30READ y
40POKE 28000+x,y
50NEXT x
60DATA 62,0,211,255,201
70END
```

As long as LOMEM is not reset below 28005, I can use the following instruction to send a value (e.g. x) to the sound chip:

```
POKE 28001,x:CALL 28000
```

The sound processor has 4 voices. The first three are normal TONE voices and the fourth is a WHITE noise voice which can be used to create sound effects. We will not get into the generation of white noise as it is quite complex and subjective. The best way to find out how white noise works is to experiment with it yourself. All four voices can be active simultaneously which can create quite spectacular effects.

Since all sound instructions are sent through the same port, we need a convention which will tell the processor which voice we wish to address. In order to generate a complete sound, three bytes are required. The first, defines the voice and volume, the second the fine note frequency, and the third the coarse note frequency. Following is the range of values for each voice:

	1	2	3	4
fine frequency	128-143	160-175	192-207	224-239
voice volume	144-159	176-191	208-223	240-255

The coarse frequency values range from 0 to 63; actually, values from 64 to 127 are identical to those from 0 to 63. The coarse frequency value is always sent to the last selected voice. It is therefore essential to send the fine frequency value before sending the coarse one.

The volume instruction has a range from 0 to 15 from the base value. It is actually an attenuation value so that 0 is the loudest and 15 is off. Thus a value of 144 sets voice 1 to maximum volume and a value of 159 turns it off. To start putting this into perspective, run the following program which installs the driver and varies first the volume of voice 1 and concludes by changing the frequency:

```
10LOMEM:28005
20FOR x=0 to 4
30READ y
40POKE 28000+x,y
50NEXT x
60DATA 62,0,211,255,201
70REM driver installed let's get to work
80send a note to voice 1
90POKE 28001,128:CALL 28000:POKE 28001,25:CALL 28000
100for x=144 to 159
110POKE 28001,x:CALL 28000:REM send the volume
120PRINT "Volume ";x-144:REM show the relative volume
130FOR w= 1 to 1000:NEXT:REM wait a bit
140NEXT x
150PRINT "now to vary the fine frequency"
160PRINT "listen very carefully"
170POKE 28001,144:CALL 28000:REM reset to max volume
180for x=128 to 143
190POKE 28001,x:CALL 28000:REM send the frequency
200PRINT "Frequency bias ";x-128:REM show the relative value
210FOR w= 1 to 1000:NEXT:REM wait a bit
220NEXT x
230PRINT"Now for the coarse frequency"
240for x=0 to 63
250POKE 28001,x:CALL 28000:REM send the frequency
260PRINT "Frequency bias ";x:REM show the value
270FOR w= 1 to 1000:NEXT:REM wait a bit
280NEXT x
290POKE 28001,159:CALL 28000:REM sound off
```

Note in the last section (starting at 230) that I did not send a FINE FREQUENCY instruction. Since the last section had sent one, it was not required. If the last instruction had been a volume setting, or an instruction to another voice, a FINE FREQUENCY instruction would have been required.

You probably noticed that as the coarse frequency values went up, the sound frequency went down. This is because we are actually modifying the wave length. Since we are more comfortable talking in terms of frequency, I will continue along those lines. Sound frequencies values are based on ADAM's clock which has a frequency of 3,597,000. This value must be divided by 32 to obtain the base frequency. Thus the base is 112,406.25 but for precision should be expressed as a variable e.g. bf=3597000/32. This value must then be divided by the frequency of the sound we wish to generate. Following is a musical scale and corresponding values:

OCTAVE	NOTE	FREQBASE	COARSE	FINE
2	A	110	102	26314
2	A#	116.64	96	360 3
2	B	123.47	91	05614
3	C	130.81	85	95311

3C# 138.59 8115011
3D 146.83 7664714
3D# 155.57 72345 3
3E 164.82 6824210
3F 174.62 64440 4
3F# 185 60838 0
3G 196 5743514
3G# 207.65 5413313
3A 220 5113115
3A# 233.08 48230 2
3B 246.94 45528 7
4C 261.62 4302614
4C# 277.18 40625 6
4D 293.66 3832315
4D# 311.13 36122 9
4E 329.63 34121 5
4F 349.23 32220 2
4F# 370 30419 0
4G 392 2871715
4G# 415.3 2711615
4A 440 2551515
4A# 466.16 24115 1
4B 493.88 22814 4
5C 523.24 21513 6
5C# 554.36 2031211
5D 587.32 1911115
5D# 622.26 18111 5
5E 659.26 1711011
5F 698.46 16110 1
5F# 740 152 9 8
5G 784 143 815
5G# 830.6 135 8 7
5A 880 128 8 0
5A# 932.32 121 7 9
5B 987.76 114 7 2
6C1046.5 107 611
6C#1108.72 101 6 5
6D1174.64 96 6 0
6D#1244.52 90 510
6E1318.52 85 5 5
6F1396.92 80 5 0
6F#1480 76 412
6G1568 72 4 8
6G#1661.2 68 4 4
6A1760 64 4 0
6A#1864.64 60 312
6B1975.52 57 3 9
7C2093 54 3 6
7C#2217.44 51 3 3
7D2349.38 48 3 0
7D#2845.04 45 213
7E2637.04 43 211
7F2793.84 40 2 8
7F#2960 38 2 6
7G3136 36 2 4
7G#3322.4 34 2 2
7A3520.4 32 2 0
7A#3729.28 30 114
7B3951.04 28 112
8C4186 27 111

As you can see from the table, there is little difference in the values for the higher octaves. This

makes it difficult to accurately interpret notes in the higher ranges. Now that we have the base values in column 4, what do we do with them? Let's work with C in the 6th octave which has a value of 107. We must split it up into COARSE and FINE values. Since the fine value is in the range of 0-15, we need to decompose the note value into MODULUS 16. Dividing 107 by 16 we get 6 and a remainder of 11. Thus 6 is the coarse value and 11 is the fine value. To play the C note on voice 1, we send the three following bytes to the sound chip:

128+0+11=139128 offset+voice 0 frequency+fine frequency
6coarse note value
128+16+0=144128 offset+voice 1 volume+attenuation value

to send the same note to voice 2:

128+32+11=1716128+48+0=176

The table of notes includes all the coarse and fine values for each note. Thus you can extract the individual offsets from the table or use the base values with the following formula:

coarse=int(base/16)
fine=base-coarse*16

The white noise voice (voice 4) is manipulated in a slightly different way. Values from 240 to 255 manipulate the sound volume in the same manner as the other voices. The noise frequency values are quite different.

224buzzing sound high pitch
225buzzing sound medium pitch
226buzzing sound low pitch
227buzzing sound varied by voice 3
228white noise high pitch
229white noise medium pitch
230white noise low pitch
231white noise varied by voice 3

For an easy noise sound, you can use the first three values of each set and a corresponding volume setting. If you are already using voice three for something else, these are the only noises you can create. If voice 3 is not in use, you can expand your capabilities by using 227 or 231 and follow with a COARSE and FINE sound value to voice 3. If you don't want voice 3 to generate any sound, make sure you turn its volume off (223). As you vary the pitch of voice three you will vary the pitch of the white noise sound. You can use this technique to create very complex sounds from explosions to gunfire, motors, spaceships, and many more. Your creativity and patience are virtually the only limitation.

After creating any kind of sound, it is important to turn the volume off. You can leave the pitch values as they are and just turn the volume back on to create the same sound. If you have created a complex sound involving all 4 voices, it may be tedious to turn all 4 voices off one at a time. Fortunately the EOS already has such a routine. Just CALL 64851 (FD53). If you are generating sound from a BASIC program that might be aborted (or generate an error), you should put in an error trap (ONERR) which branches to an exit routine that turns off the sound. There are few things more annoying than breaking out of a program with the noise buzzing away and no apparent way of turning it off.

MORE ON MACHINE LANGUAGE ROUTINES

In SMARTBASIC, any PROTECTED RAM area may be used to write machine language routines. But why machine language? Because there are some operations which simply cannot be performed by standard BASIC commands or functions. Additionally, some processor intensive tasks will execute very slowly in BASIC; machine language is one way to speed them up.

In this chapter, we will look at 3 ways of accessing machine language routines from SMARTBASIC. Although source code examples will be supplied, the purpose of this article is NOT to explain Z-80 operation codes and their use -- it is intended for those who know something about ML and want to incorporate it into BASIC programs.

CALL is the simplest function. It takes care of housekeeping operations such as saving program pointers prior to executing your machine language routine. It is the programmer's responsibility, however to preserve the stack or to set up a local stack. In most cases, BASIC's enormous stack should be sufficient.

You may have seen CALL instructions used in sound generation, disk copy programs, and sprite animation. These are three examples of functions which cannot be performed any other way. Here is a sound generation routine:

```
LDA,0;put value in A
OUT(255),A;send it
RET;return to BASIC
```

In machine language, this would look like: 62 0 211 255
201. Simple enough now we must POKE it in memory:

```
10 LOMEM: 27417: REM make room for it
20 for x=0 to 4:REM read 5 values
30 READ y
40 POKE 27407+x,y: REM put in memory
50 NEXT x
60 DATA 62,0,211,255,201
```

Now if I "CALL 27407", a 0 will be sent to the sound port. If I want to send any other value, I can POKE it into memory at 27408 (where the 0 is) and CALL 27407 again. Refer to other articles on sound generation for the specific information required to create notes or sound effects.

READ/WRITE block routines have been around for some time but all the ones I have seen lack one essential element: they neglect to check for errors. When the EOS performs a read/write operation, the error code is returned in register A. Since we have no direct access to registers, we must devise another approach; here's my routine which I install at 27420:

```
XORA;clear accumulator
LD(27419),A ;reset error
LDA,0;device
LDBC,0;high block number
LDDE,0;low block number
LDHL,0;RAM address
CALL64755;read block
RETZ;no error
LD(27419),A ;set error
RET;exit
```

This routine starts by clearing the error then load A with the device, DE with the block number (BC must always be 0) and HL with the address where the information will be read or written. 64755 is the address of the READ block routine, 64758 is the write block routine. If there is no error, the

program just returns and the error code in 27419 is zero. When there is an error, the error code will appear in 27419. Following are the addresses for the parameters:

27425device number 4,5,8,24
27430-31block number lo-hi
27433-34RAM address
27436243 for read 246 for write

Following is the entire data stream to POKE in the machine language routine starting at 27420:

175 50 27 107 62 0 1 0 0 17 0 0 33 0 0 205 243 252
200 50 27 107 201

The last byte (201) should be at 27442.

To use this routine, request or determine the block number, memory address, and device number, select read/write, and CALL 27420. Following is part of a subroutine to do that:

```
99REM R/W error check subroutine
100...poke block, memory, device, etc
...
150CALL 27420
160e=PEEK(27419)
170IF e=0 then return:  REM all ok
180PRINT "Error ";e
190STOP
```

When using this routine, the program will abort when the EOS reports an error. If you want to get more sophisticated, following are some of the error codes:

1,2,3device unavailable
5no file
6file exists
11file too big
12directory full
13tape/disk full
15rename error
16delete error
17range error
21bad device status
24bad directory on medium

Now for something more complex: SPRITE animation! Sprites are managed in 2 tables which the programmer must maintain in RAM:

The first has 32 segments of 8 bytes (256 total) which represent the sprite shape. The 8 bytes represent a horizontal bitmap (on off) of each of the 8 lines of a sprite shape. Thus if sprite number 0 is a circle, you could have the following definition:

00011000 24
00111100 60
01100110102
11000011195
11000011195
01100110102
00111100 60

00011000 24

The next table has 32 elements of 4 bytes which represent the first byte of each set of 4 is the Y coordinate, the second is the X coordinate, the third is the sprite number, and the fourth is the sprite colour. The sprite number may be any value from 0 to 31; the first sprite does not need to be 0. Furthermore, you can use the same sprite definition to place the same shape in different colours on different parts of the screen. You are limited, however to 32 sprite definitions (shapes) and 32 sprite attributes (location and colour).

Sprites can be displayed in 4 fashions:

SIZEVALUE
8x8E0H (224)
8x8 doubleE1H
16x16E2H
16x16 doubleE3H

Following is a machine language routine which updates all 32 sprite positions at once:

```
LDB,1;register
LDC,0E0H;use 8x8 sprites
CALL64800;update vdp
LDA,1;shape definitions
LDIY,32;do all sprites
LDDE,0;offset in table
LDHL,28256 ;my def's are here
CALL64812;update vram
LDA,0;position table
LDIY,32;do all sprites
LDDE,0;offset in table
LDHL,28512 ;my table is here
CALL64812;update vram
RET
```

This routine can be POKED anywhere in memory; I use 28000
with the following data:

```
6 1 14 224 205 32 253 62 1 253 33 32 0 17 0 0
33 96 110 205 44 253 62 1 253 33 32 0 17 0 0
33 96 111 205 44 253 201
```

The last byte should be at 28037.

Three user parameters must be supplied: the size and the
location of the 2 user tables:

```
28003  sprite size E0 E1 E2 E3
28017  shape definition (lo-hi)
28032  position table (lo-hi)
```

The VRAM write routine can be indexed in order to update only one sprite at a time but this would require several modifications to the routine on every CALL. Once you get this installed, you will find that it is fast enough that you don't need to bother for most applications; you would probably spend more time calculating offsets.

In the examples above, I talked about references to memory addresses as lo-hi. This means that you

must POKE 2 values which represent a number from 0-65535. Following is the easiest method to make this calculation:

```
100 x=28017:  REM POKE it here
110 y=28256:  REM POKE this value
120 GOSUB 900
...
899 REM routine to POKE y into x
900 POKE x+1,y/256:  REM high byte
910 POKE x,y-256*PEEK(x+1):  REM low byte
920 RETURN
```

Note in line 900 that y/256 does not necessarily result in an integer value; the POKE command, however, converts all values to integer prior to placing them in memory. Thus extracting the POKed value by using PEEK in line 910 correctly calculates the integer value of the high byte.

If you intend to use complex machine language routines, you may occasionally want to print something to the screen. Here you can make use of 2 routines which already exist in SMARTBASIC:

PRINT CHARACTER IN A resides at 11994. Thus if you want to print a question mark, you simply do:

```
LDA,'?'
CALL11994
```

The advantage of the print routine is that it will perform word wraps and screen scrolls when required.

If you want to print a message, you can use the length_encoded routine at 12110:

```
LDHL,MESSAGE
CALL12110
.....
MESSAGE:
DB14;length of message
DB13;a carriage return
DB'Guy Cousineau'
```

If you want to get USER input into your routines, you can use the EOS read keyboard routine located at 64620; it returns a character in register A. Note that this routine waits until a character is pressed:

```
CALL64620;get character
CP3;is it ^C
RETZ;yes, abort
CP'y'; is it YES
JPz,YES;process yes answer
.....
```

Thus you could use a USR function to get a routine started and it could prompt the programmer/player for the required parameters.

If you plan on creating complex machine language routines, you may wonder how you will ever accurately determine the POKE values. If you have CP/M or TDOS, you can save a lot of work by using a Z-80 assembler in CP/M. Then you can use the resulting PRN or HEX file to determine the POKE values and critical addresses in your routines.

The following program may be even more useful. Start by writing your routine in CP/M and assemble to a HEX file. Then use CPM.COM (CP/M) or FC.COM (TDOS) to convert the HEX file to EOS format. The next step is to run this program which will POKE your routine in memory for you:

```
100 INPUT "file to assemble "; f$
110 INPUT "drive number "; d
200 ONERR GOTO 500
210 ?  CHR$(4);"open "; f$; ",d"; d
220 ?  CHR$(4);"read "; f$
229 REM extract the load address
230 INPUT w$:  h$=MID$(w$, 4, 2):GOSUB 300
235 w=v*256:  h$=MID$(w$, 6, 2):GOSUB 300:  q=v+w
240 w$=" "+w$:  p=q:?  "first byte at", p
250 FOR x=11 TO LEN(w$)-3 STEP 2
260 h$=MID$(w$, x, 2):GOSUB 300:  REM get a value
265 POKE p, v:  p=p+1:NEXT
270 INPUT w$:IF MID$(w$, 3, 6)<>"000000" GOTO 250
280 GOTO 500 :REM end of file
300 a=ASC(LEFT$(h$, 1))-48:  REM high nibble
305 b=ASC(RIGHT$(h$, 1))-48:  REM low nibble
310 v=(a-7*(a>9))*16+b-7*(b>9):RETURN
500 CLRERR:?:?  CHR$(4);"close "; f$
510 p=p-1:?  "last byte at ", p
540 ?:?  " BSAVE     ,A"; q; ",L"; p-q+1
550 ?  CHR$(160); :? CHR$(160);
```

The last thing the program does before exiting is to calculate the length of your routine and supply you with a BSAVE instruction line. Just scroll past BSAVE, enter a file name, scroll to the end of the line and press RETURN. Now your ML routine is saved as a file which can be quickly reloaded via a BLOAD command.

When creating your routines, be sure to use the assembler ORG directive to set the destination address of your routine above 27407

Once you start experimenting with machine language routines, you may quickly discover the benefits. Sort routines, for example, will run up to 100 times faster in machine language. Should you have any questions about machine language programming, you may contact me.

This concludes my analysis of SMARTBASIC. Should any major topics have been omitted, please let me know. Also, please advise if you would like additional information on any of the topics covered in this series. For a complete set of articles (CP/M or MS-DOS text) or this document (WordPerfect 5.1), please send two single sided formatted disks or one double sided disk, or one DDP for the CP/M information, or one MS-DOS disk to the undersigned.

Guy Cousineau
1059 Hindley Street
OTTAWA Canada
K2B 5L9