Notice how straightforward the logic is, even though we are still checking for operator errors. Any entry other than one of the six allowed characters is rejected. If changing the row or column number puts it out of the allowed range, then program logic simply asks for new row and column numbers. (The CBM 8000 window and scrolling functions are not very useful in this example since we want to scroll left and right, as well as up and down.)

An untidy aspect of the program shown above is the fact that, following an out of range *row* number, only a new row is allowed to be entered; this results from the GOTO 100 on lines 210 and 230. Following an out of range *column* number the GOTO 70 on line 300 allows new column *and* row numbers to be entered (since in the main body of the program, column number entry precedes row number entry). Can you rewrite the program to get rid of this small untidiness (select whether only the row or column will be reentered, or if both the row *and* column will be reentered when either is out of range)?

Another undesirable feature of the display program is the time taken to fill the array X%. This has nothing to do with the display itself, but in many programs such delays are likely to occur. An operator may well assume that the computer is not working properly. Whenever such periods of inactivity are encountered it is a good idea to display a prominent message telling the operator that the computer is working, and to please wait. This is easily done. You simply precede the computation statements with an appropriate PRINT statement. In our case the following PRINT statement could be used:

```
15 PRINT "PLEASE WAIT WHILE I FILL THE ARRAY WITH DATA"
```

Our program takes great care to terminate the display on the 39th column of the display, rather than the 40th and last column. **When using a CBM computer with a 40-column display, it is not wise to run displays out to the 40th column. You will run afoul of the wrap around logic whereby lines that are more than 40 characters long automatically continue on the next line.** You are best off not tangling with the display formatting nightmare that can result from carriage returns generated as part of line continuation interacting with your own formatting carriage returns.

40-Column Screen Wrap Around Logic. The following paragraphs explain how 40-column wrap around logic works.

When the cursor is on any 40-character screen line, the CBM computer assumes that it is a 39-character line until a character has been displayed in the 40th character position; then the CBM computer assumes it is in the first half of a 79-character line. If a character has been displayed in the 40th column of the preceding line (i.e., the cursor has moved to the next line), then the CBM computer assumes it is in the second half of a 79-character line.

When a program encounters a carriage return, it executes a carriage return to the next logical line. When the CBM thinks it is in the first half of a 79-character line (a character has been displayed at the 40th character position) and it executes a carriage return, it moves the cursor to the next logical line, which is two display lines below.

If you POKE into the 40th character position of a 40-character display then the computer does not assume a 79-character line. This can be done using the statement:

```
POKE 32767+(L-1)*40,ASC(CH$)
```

where:

L	is the line number
CH$	is the POKEd character

If you have a 40-column display, then as an exercise it is worth modifying the complete table display program so that it does go out to the 40th column. To do this you

must change the TABs on line 30 and line 1010 from 9 to 10; the TAB on line 1050 must change from 13 to 14, the TAB on line 1130 goes from 2 to 3. Now try running the program; the columns of numbers line up, but you have too many carriage returns and they force the top of the display to scroll off the screen. Now try eliminating the extra carriage returns and generating the correct display. This is a very difficult programming task.

MATHEMATICAL PROGRAMMING

CBM computers can add, subtract, multiply, and divide with full accuracy using numbers that have up to nine digits. Numbers with more digits have to be rounded off to nine digits. Thus 123456789.12 is rounded to 123456789. Although this limit poses no problem in many applications, business and scientific applications can require more digits of accuracy. The CBM cannot keep track of dollars and cents (to the nearest cent) for amounts over $9,999,999.99, for example.

Two programming methods can overcome the CBM computer's numeric accuracy limitations. The first method uses numeric strings. The second method uses multiple integer math, where a large number is separated into smaller segments, and each segment is handled separately.

ADDITION

Numeric string and multiple integer techniques can both be used to add integer numbers that have more than nine digits. The *augend* is the first number in the equation. The *addend* is the second number. The addend is added to the augend.

Addition using Numeric Strings

The steps involved are:

1. Input the augend and addend as two positive numeric strings.
2. Right justify the strings.
3. Add the corresponding digits of the strings separately, including carry.
4. Concatenate the answer into a one-string result.
5. Print the answer string.

Let us examine each step in turn:

Step 1: Input the augend and addend as positive numeric strings using an INPUT statement.

Screen Display

```
10 PRINT"■***ADDITION***":PRINT
20 INPUT A$,B$

RUN

***ADDITION***

?123456789123456
??57943572
```

Representation of Memory Contents

```
A$ [1][2][3][4][5][6][7][8][9][0][1][2][3][4][5][6]
B$ [5][7][9][4][3][5][7][2]
```

A$ is the augend and B$ is the addend. The INPUT statement allows either to exceed the 9-digit numeric length limit. For simplicity we will allow only positive integer numbers to be input. Once you are familiar with the basic concepts of the addition program, you should experiment and alter the program to accommodate negative and fractional numbers.

Step 2: Right justify the strings. Before performing arithmetic operations, the numbers should be right-justified, because in BASIC alphabetic and numeric strings are automatically left-justified. If the contents of numeric strings are added without first being right-justified, the answer will be incorrect, as shown below:

Left Justified - Incorrect	Right Justified - Correct
1234567890123456	1234567890123456
+57943572	+ 57943572
7028925090123456	1234565948067028

The following statements right-justify the shorter of the two numeric strings A$ and B$. The shorter string is filled with leading zeros until it equals the length of the longer string. X is assigned the length of A$. Y is assigned the length of B$:

```
30 BLANK$="
40 X=LEN(A$):Y=LEN(B$)
50 IF X<Y THEN A$=LEFT$(BLANK$,Y-X)+A$
60 IF Y<X THEN B$=LEFT$(BLANK$,X-Y)+B$
```

BLANK$ on line 30 is a buffer string that is used to fill the shorter numeric string with blanks. BLANK$ has 16 blank spaces, since we are going to simplify our problem by imposing a 16-digit limit on the size of numbers.

Statements on lines 50 and 60 use the LEN function to compare X (the length of A$) to Y (the length of B$), and subtract the length of the smaller string from the length of the larger string. In our example B$ is shorter than A$, so the length of B$ is subtracted from the length of A$.

```
60 IF Y<X THEN B$=LEFT$(BLANK$,X-Y)+B$
```

Length of smaller string subtracted from length of larger string

If the length of A$ is 16 digits and the length of B$ is eight digits, the difference is eight digits:

A$	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6	X = 16	X - Y = 8
B$	5 7 9 4 3 5 7 2	Y = 8	16 - 8 = 8

The number of blanks concatenated onto the front of B$ is the difference between the two lengths. Since the difference is eight, eight blanks are taken from BLANK$ to fill the shorter string. Blanks are added to the front of the shorter string B$ with the following statement:

```
LEFT$(BLANK$(X-Y)+B$
```

The procedure is as follows:

```
B$=LEFT$(BLANK$,X-Y)                          +B$
B$=LEFT$(BLANK$,16-8)                         +B$
B$=LEFT$(BLANK$,8)                            +B$
B$=LEFT$(ø ø ø ø ø ø ø ø ø ø ø ø ø ø ø ø ,8)   +B$
B$=ø ø ø ø ø ø ø ø                             + 5 7 9 4 3 5 7 2
B$=ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2

A$=1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6            B$=ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2
        16 digits                                      16 digits
```

Step 3: Add the corresponding digits of the strings. At first glance, you might assume that A$ and B$ can now be added using the following statement:

```
C$=A$+B$
```

This is incorrect. When a plus sign is used with strings they are not added, but are concatenated:

```
C$=A$+B$
C$=1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6  +  ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2
C$=1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2
```

We want to add the digits in the strings, not concatenate the strings. **To add the contents of numeric strings, each digit must be extracted separately from the string, converted into a numeric digit, then added to one digit from the other string.** This is done using the two string functions VAL and MID$.

```
1020 FOR I=LEN(A$) TO 1 STEP-1
1030 A=VAL(MID$(A$,I,1))
1050 B=VAL(MID$(B$,I,1))
1100 NEXT I
```

A is the digit extracted from A$. B is the digit extracted from B$. I is a counter initialized to the length of the INPUT strings (either A$ or B$ may be used). With each FOR-NEXT loop iteration, the value of I is decremented by 1. As I decrements, it allows the string contents to be extracted one by one, right to left, using the MID$ function:

```
1          MID$(B$,I,1)
16  ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2
15  ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2
14  ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2
13  ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2
12  ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2
11  ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2
10  ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2
 9  ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2
 8  ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2
 7  ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2
 6  ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2
 5  ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2
 4  ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2
 3  ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2
 2  ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2
 1  ø ø ø ø ø ø ø ø 5 7 9 4 3 5 7 2
```

The VAL function converts each extracted string literal into a numeric value:

When I = 16,

```
B=VAL(MID$(B$,16,1))
B=VAL($ 0 0 0 0 0 0 0 0 0 5 7 9 4 3 5 7 2 )
B= 0 2
```

When I = 15,

```
B=VAL(MID$(B$,15,1)) . . .
```

After both numeric string digits have been converted into an integer, they are added and the sum is returned in C$. Here are the necessary program steps:

```
1000 N=1                                  Initialize string pointer N.
1010 D=0                                  Initialize carry value.
1020 FOR I=LEN(A$) TO 1 STEP -1           Initialize decrement counter I.
1030 A=VAL(MID$(A$,I,1))                  Extract digits separately. Convert
                                            to non-string numeric.
1040 A=A+D:D=0                            Add tens value from carry (D) to A.
1050 B=VAL(MID$(B$,I,1))                  Extract digits separately. Convert
                                            to non-string numeric.
1060 C=A+B                                Add extracted digits of A$ and B$.
1070 IF C>=10 THEN D=1                    Carry tens value into D if C>=10.
1080 IF D=1 AND I=1 THEN N=2
1090 C$=RIGHT$(STR$(C),N)+C$              Link sums into string answer.
1100 NEXT I
```

Variable D is initialized to zero at line 1010; D is then used as a carry value in lines 1040, 1070, and 1080. During addition, if the value of C is greater than or equal to 10, a tens value is *carried over* to the next left position. The tens value carried over is stored in D:

```
  1 2 3 +1 4 +1 5 +1 6  1 7 +1 8 9 0 1 2
+         5   7   9   4 3 5 7 2
  1 2 3   5   1   4   7   3 2 5 8 4
```

If C is greater than or equal to 10, the carry variable D is incremented to 1 at line 1070; otherwise it remains 0:

```
1070 IF C>=10 THEN D=1

A   0 6
+B  0 9
C   0 15  ——→ 15>=10 → D 0 1

or

A   0 3
+B  0 0
C   0 3   ——→ 3<10 → D 0 0 (no change)
```

D will be either 0 or 1, but never greater than 1, because the maximum possible sum of any two single-digit numbers is 18, thus the maximum tens value that can be carried over is 1.

To prevent losing the carry in D, line 1040 resets the value of A to A + D on the next loop iteration:

```
1040 A=A+D:D=0
```

If this statement were omitted, the carry would never be carried out, and the value of A would be incorrect. When D is added to A, D is reset to 0 in preparation for the next loop iteration.

Step 4: Link the individual sums (C) and convert the total sum into a string. Just as the augend and addend were entered as strings to avoid the 9-digit length limit the sums must be converted back into a string to avoid the length limit.

Line 1090 links the individual sums of C and converts the final answer back into string form.

The STR$(C) function converts C into a string. The RIGHT$ function extracts the rightmost N characters from STR$(C). N is set to 1 at line 1000 to indicate that we want only the rightmost character to be extracted; the leftmost character of C is unnecessary because it is the sign value ("Ø" if positive and "−" if negative) and would be concatenated between each number of C$ if we did not exclude it.

```
1000 N=1
     N[Ø1]
1060 C=A+B
     C[Ø8] = A[Ø6] + B[Ø2]
1090 C$=RIGHT$(STR$(C),N)+C$
     C$=RIGHT$(STR$(C),1)+C$
     C$=RIGHT$([Ø8],1)+C$
     C$=[8] + C$
```

Even if C is a two-digit number, only the rightmost digit is concatenated onto C$. The tens value has already been assigned to D and will be added during the next loop iteration.

N is set to 2 to include the last carry only if D=1 and I=1 (signaling a carry on the last loop iteration). This is important, because if both conditions are true the loop will *not* iterate again to add D's carry into A in line 1040, thereby losing the last carry value in D. By setting N to 2 on the last loop iteration, both digits of C are included in C$, and the last carry over is not lost.

```
1070 IF C>=10 THEN D=1
        C[Ø12]>=10    D[Ø1]
1080 IF D=1 AND I=1 THEN N=2
        D[Ø1]      I[Ø1]       N[Ø2]
1090 C$=RIGHT$(STR$(C),N)+C$
     C$=RIGHT$([Ø12],2)+C$
     C$=[12]+C$
     C$=[12XXXXXXXX]
```

The entire FOR-NEXT loop routine at lines 1020 through 1100 does as follows:

1. It extracts individual digits from a numeric string and assigns numeric values to them (statements 1030, 1050).

2. The digits from both strings are added together one digit at a time (statement 1060) and checked for a carry value (statement 1070). The carry is added to A in the next column (line 1040).
3. The individual sums are then linked and converted back into a numeric string (line 1090).

Step 5: Display the answer string. To complete this addition routine, the input and length test statements are inserted at the beginning of the FOR-NEXT loop (statements 10 to 1010). PRINT and CLEAR statements are added (statements 1110 to 1130). The final program now reads as follows:

```
10 PRINT"□***ADDITITON***":PRINT                    Clear screen
20 INPUT A$,B$                                      Input numeric strings
30 BLANK$="                                "
40 X=LEN(A$):Y=LEN(B$)
50 IF X<Y THEN A$=LEFT$(BLANK$,Y-X)+A$  }
60 IF Y<X THEN B$=LEFT$(BLANK$,X-Y)+B$  }  Right justify strings
1000 N=1
1010 D=0
1020 FOR I=LEN(A$) TO 1 STEP-1
1030 A=VAL(MID$(A$,I,1))
1040 A=A+D:D=0
1050 B=VAL(MID$(B$,I,1))
1060 C=A+B                                          Addition loop
1070 IF C>=10 THEN D=1
1080 IF D=1 AND I=1 THEN N=2
1090 C$=RIGHT$(STR$(C),N)+C$
1100 NEXT I
1110 PRINT:PRINT"ANSWER= ";C$                       Print C$
1120 C$="":PRINT:GOTO 20                            Clear C$
1130 END
```

Two sample runs of the program give the following output:

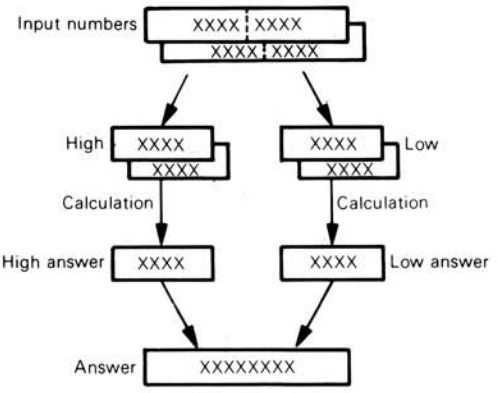
```
***ADDITION***

?12345
??579

ANSWER=  12924

?1234567890123456
??57943572

ANSWER=  1234567948067028
```

This addition routine overcomes the 9-digit numeric length limit. Try modifying this program to receive inputs as dollars and cents, and to display results in the same format.

Multiple Integer Addition

Another way to overcome the 9-digit length limit during addition is to use multiple integer addition.

Multiple integer math reorganizes a large number into smaller segments. Each segment is handled independently. The individual answers are joined together into one final answer, as follows:



The steps involved in multiple integer addition are as follows:

1. Input the augend and addend as two positive numeric strings.
2. Divide the number into two equal high and low parts.
3. Separately calculate the sums of the high-order and low-order parts.
4. Concatenate the sums into one answer string.
5. Display the answer string.

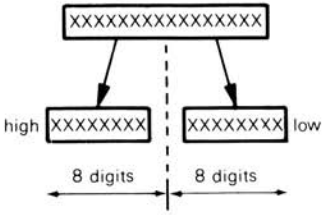
Step 1: Input the augend and addend as two positive numeric strings:

```
10 PRINT"□***MULTIPLE INTEGER ADDITION***":PRINT
20 INPUT A$,B$

RUN

***MULTIPLE INTEGER ADDITION***

?1234567890123456
??57943572
```

A$ is the augend and B$ is the addend. The numbers are input as numeric strings because: 1) the numeric length limit is avoided, and 2) string functions can be used to divide the numbers into smaller segments.

Step 2: Determine the maximum length of numeric input, and the number of segments into which the numeric input must be divided. For example, if the maximum length of numeric input is 16 digits, numbers must be divided into two segments, with a maximum of eight digits per segment.

To keep our sample program simple, the maximum input length is assumed to be 16 digits. Input is divided into high and low segments of eight digits each.



First we must determine which input string is longer. The lengths of A$ and B$ are assigned to variables X and Y respectively.

```
1000 X=LEN(A$):Y=LEN(B$)
```

Next, the lengths are compared. If X>Y (length of A$ is larger than length of B$) then variable F, the divider variable, is set to one-half of X. But if X<Y, then F is set to one-half of Y.

```
1002 IF X>Y THEN F=X/2:GOTO 1006
1004 F=Y/2
```

Here is another method of assigning a value to F:

```
1002 F=Y/2:IF X>Y THEN F=X/2
```

In this example, A$="1234567890123456" and B$="57943572." Let us run this through:

```
1000 X=LEN(A$):Y=LEN(B$)
     X = 16          Y = 8
1002 IF X>Y THEN F=X/2:GOTO 1006
     16>8 true statement. therefore
          F = 16/2
          F = 8
          program continues at line 1006
```

Once the value of F is set, the program continues at line 1006. The statement on line 1006 looks for a fractional value of F. If F is larger than its integer value, then F is assigned its integer value, plus 1. This rounds F up to the nearest integer. For example, if the value of F is 7.5, the statement on line 1006 rounds it up to 8:

```
1006 IF F>INT(F) THEN F=INT(F)+1
     If 7.5>7 then F = 7+1
                   F = 8
```

To obtain the high (H) and low (L) parts of the sum of A$ and B$, use the following statements:

```
1000 X=LEN(A$):Y=LEN(B$)
1002 IF X>Y THEN F=X/2:GOTO 1006
1004 F=Y/2
1006 IF F>INT(F) THEN F=INT(F)+1
1010 IF X<=F THEN AH=0:AL=VAL(A$):GOTO 1040
1020 AH=VAL(LEFT$(A$,X-F))
1030 AL=VAL(RIGHT$(A$,F))
1040 IF Y<=F THEN BH=0:BL=VAL(B$):GOTO 1070
1050 BH=VAL(LEFT$(B$,Y-F))
1060 BL=VAL(RIGHT$(B$,F))
```

Statements 1010 and 1040 compare the string lengths with the divider F, which in this case is 8. If the string is shorter than eight, AH (or BH) is assigned a zero value, leaving only AL (or BL) equal to A$ or B$. If the string is longer than eight, it must be divided into high and low segments. AH or BH, the high segments, are assigned the value of the leftmost LEN(X or Y), minus eight digits, at 1020 and 1050.

```
1020 AH=VAL(LEFT$(A$,X-F))
     AH=VAL(LEFT$(A$,16-8))
     AH=VAL(LEFT$("1234567890123456",8))
     AH=VAL("12345678")
     AH=12345678
```

To obtain AL, the rightmost eight digits are extracted from A$:

```
1030 AL=VAL(RIGHT$(A$,F))
     AL=VAL(RIGHT$("1234567890123456",8))
     AL=VAL("90123456")
     AL=90123456
```

The same procedure is used to extract BH and BL. Notice that *the VAL function converts the strings into numbers.*

Step 3: Once the large strings are divided into segments small enough for the CBM computer to handle, addition can begin. With multiple integer addition, you **add corresponding groups of numbers.** AH and BH are added. AL and BL are added. When a number is handled as a group of digits and not as a numeric string, the addition of each number does not have to be done digit by digit as with the numeric string method. The CBM computer can add *numbers,* whereas it is unable to add *numeric strings.*

```
AH  ø12345678        AL  ø90123456
+BH ø00000000        +BL ø57943572
---------------      ---------------
CH  ø12345678        CL  ø148067028
```

First, the low segments AL and BL are added using the following statement:

```
1070 CL$=STR$(AL+BL)
```

The sum of AL and BL is converted into a numeric string when assigned to CL$. It is not necessary that the sum be in string form, but it is much simpler to test for carry-over using the LEN function.

Line 1075 truncates the leading blank from the front of CL$. Remember that when a number is converted into a string the leading blank is included. We do not want this leading blank as part of CL$ when we concatenate the high and low segments together; therefore we truncate it with the MID$ function.

Line 1080 tests the length of sum CL$ against the segment length F. If the length of CL$ is greater than F, the leftmost digit is carried over and added to the sum CH$. (The value of D is equal to either 0 or 1.)

CH$ is obtained by adding AH, BH, and the carry D.

```
1070 CL$=STR$(AL+BL)
     CL$=STR$(ø90123456 + ø57943572)
     CL$=STR$(ø148067028)
     CL$=ø148067028
1075 CL$=MID$(CL$,2,LEN(CL$)-1)
     CL$=MID$(ø148067028,2,10-1)
     CL$=MID$(ø148067028,2,9)
     CL$=148067028
1080 IF LEN(CL$)>F THEN D=1
     LEN(CL$)=9   :F=8
     9>8→D=1
1090 CH$=STR$(AH+BH+D)
     CH$=STR$(ø12345678 + ø00000000 + ø1)
     CH$=STR$(ø12345679)
     CH$=ø12345679
1095 CH$=MID$(CH$,2,LEN(CH$)-1)
     CH$=MID$(ø12345679,2,10-1)
     CH$=MID$(ø12345679,2,9)
     CH$=12345679
```

Step 4: Next we **concatenate the two sums** into one answer by linking CH$ to the front of CL$. The preceding space and carry are truncated from CL$ by selecting the rightmost eight digits from that string.

```
1100 C$=CH$+RIGHT$(CL$,F)
     C$=CH$[ 0 1 2 3 4 5 6 7 9] + RIGHT$(CL$[ 0 1 4 8 0 6 7 0 2 8],8)
     C$=[ 0 1 2 3 4 5 6 7 9] + [4 8 0 6 7 0 2 8]
     C$=[ 0 1 2 3 4 5 6 7 9 4 8 0 6 7 0 2 8]
```

Step 5: Print the answer C$.

```
1110 PRINT:PRINT"ANSWER=";C$:PRINT
```

The program is now complete. This Multiple Integer Addition program accepts two positive integer numbers that can be up to 16 digits long. The numbers are divided into high and low segments of eight digits each. The high and low segments are added and the two sums are concatenated into a single string answer with a maximum length of 17 digits. This Multiple Integer Addition program allows you eight more digits than the CBM computer's maximum.

Below is the listing of the complete program with a sample run.

```
10 PRINT"□***MULTIPLE INTEGER ADDITITON***":PRINT
20 INPUT A$,B$
1000 X=LEN(A$):Y=LEN(B$)
1002 IF X>Y THEN F=X/2:GOTO 1006
1004 F=Y/2
1006 IF F>INT(F) THEN F=INT(F)+1
1010 IF X<=F THEN AH=0:AL=VAL(A$):GOTO 1040
1020 AH=VAL(LEFT$(A$,X-F))
1030 AL=VAL(RIGHT$(A$,F))
1040 IF Y<=F THEN BH=0:BL=VAL(B$):GOTO 1070
1050 BH=VAL(LEFT$(B$,Y-F))
1060 BL=VAL(RIGHT$(B$,F))
1070 CL$=STR$(AL+BL)
1075 CL$=MID$(CL$,2,LEN(CL$)-1)
1080 IF LEN(CL$)>F THEN D=1
1090 CH$=STR$(AH+BH+D)
1095 CH$=MID$(CH$,2,LEN(CH$)-1)
1100 C$=CH$+CL$
1110 PRINT:PRINT"ANSWER=";C$:PRINT
1120 AH=0:AL=0:BH=0:BL=0:D=0:CH$="":CL$="":C$="":GOTO 20
1130 END

***MULTIPLE INTEGER ADDITION***

?1234567890123456
??57943572

ANSWER= 1234567948067028
```

Try modifying this program to receive inputs and display results as dollars and cents.

SUBTRACTION

As with addition, you can subtract numbers with more than nine digits by using numeric strings, or by using multiple integer math.

Subtraction using Numeric Strings

This subtraction program contains many sections of the "Addition using Numeric Strings" program. The steps involved are as follows:

1. Input the minuend and subtrahend as two positive numeric strings.
2. Right justify the strings.
3. Determine the larger numeric string.
4. Subtract corresponding digits of the strings separately, with borrowed carries.
5. Concatenate the answer into a one-string result.
6. Eliminate leading zeros in the answer string.
7. Print the answer string.

Step 1: The first step is to **input the minuend and subtrahend** as two positive numeric strings using an INPUT statement:

```
10 PRINT"***SUBTRACTION***":PRINT
20 INPUT A$,B$

RUN

***SUBTRACTION***

?123456789012                          A$ 1 2 3 4 5 6 7 8 9 0 1 2
??57943572                             B$ 5 7 9 4 3 5 7 2
```

A$ is the *minuend* (the first or top number entered, from which another number is subtracted). B$ is the *subtrahend* (the number subtracted from the minuend).

Step 2: Align the minuend and subtrahend by right-justifying both numeric strings. This is the same as was presented in step 2 of the "Addition using Numeric Strings" program.

```
30 BLANK$="                                "
40 X=LEN(A$):Y=LEN(B$)
50 IF X<Y THEN A$=LEFT$(BLANK$,Y-X)+A$
60 IF Y<X THEN B$=LEFT$(BLANK$,X-Y)+B$
```

Step 3: For subtraction, we must **determine which numeric string has a larger value.** Although the input strings may be equal in length, their values can be quite different.

The values of A$ and B$ are compared using the VAL function in statements 65 and 70:

```
65 IF VAL(A$)=VAL(B$) THEN C$="0":GOTO 1150
70 IF VAL(A$)>VAL(B$) GOTO 1000
```

We are going to subtract B$ from A$.

If A$ is larger than B$, we have a simple subtraction problem, and the program drops to line 1000. If B$ is larger than A$, we are subtracting a larger number from a smaller number; the program prepares for a negative answer.

If the subtrahend is larger than the minuend (B$ is larger than A$), the answer will be negative. To subtract two numbers that yield a negative answer, we switch the contents of A$ and B$ so that the value of A$ is larger than B$. Subtract B$ from A$, and the difference is C$. To make C$ negative, a negative sign, "−", is concatenated onto the front of C$: C$="−"+C$.

Let us subtract 5 from 3, for example. This presents a subtraction problem where VAL(B$)>VAL(A$), or the subtrahend is larger than the minuend.

```
A$ [3]
B$ [5]
Switch A$ and B$
      A$[5]   B$[3]  →  A$[5]   B$[3]
Subtract: VAL(A$)-VAL(B$)=C$
      A$[5] - B$[3]  →  C$[2]
Convert to negative
      C$ = "-" + C$
      "-" + C$ [2]  →  C$ [-2]
Answer:
      C$ [-2]
```

The variables are switched at line 80.

```
80 X$=A$:A$=B$:B$=X$
```

Program Statement	Memory		
	X$	A$	B$
⋮	0	3	5
X$=A$	3	3	5
A$=B$	3	5	5
B$=X$	3	5	3

X$ acts as a storage string. Without X$, the original contents of A$ would be written over and the contents of B$ would be written back into itself:

Program Statement	Memory		
	A$	B$	
⋮	3	5	
A$=B$	5	5	Incorrect
B$=A$	5	5	

Later in the program we will need to know if the variables have been switched. We therefore set a marker to signal that A$ and B$ have been switched. Use variable S for this: S remains 0 if the variables have not been switched. If the variables are switched, set S=1. Line 90 sets S=1 if the values of A$ and B$ have been switched.

```
90 S=1
```

Remember that after the strings are properly switched, a value of 1 is assigned to S to signal that the numbers have been switched and a negative answer is needed. The negative answer is obtained by concatenating a negative sign to the front of the answer before it is printed. This occurs at statement 1140.

```
1140 IF S=1 THEN C$="-"+C$
```

Step 4: Whether the final answer is negative or positive, the value of A$ is now larger than B$. We can now **perform simple subtraction** at lines 1000 and 1080. The routine is taken directly from lines 1020 to 1100, step 3 of the "Addition using Numeric Strings" program, because the digits are extracted from the strings in the same manner. However, at line 1050, the carry variable D is now used as a "borrow" variable. If (A−B)<0, then a tens digit must be borrowed from the adjacent left column, increasing the value of A by 10. D is set to −1 because a "1" is being borrowed, thereby decreasing the value of the adjacent left column. The result is C:

```
1000 REM**SUBTRACTION ROUTINE**
1010 FOR I=LEN(A$) TO 1 STEP-1
1020 A=VAL(MID$(A$,I,1))
1030 A=A+D:D=0
1040 B=VAL(MID$(B$,I,1))
1050 IF (A-B)<0 THEN D=-1:A=A+10
1060 C=A-B
```

					+10	+10	+10			~~+10~~ +9	+10	
A	1	2	3	3 ~~4~~	4 ~~5~~	5 ~~0~~	7	8	8 ~~9~~	~~0~~	1	2
−B	0	0	0	0	5	7	9	4	3	5	7	2
C	1	2	3	3	9	8	8	4	5	4	4	0

Step 5: Concatenate the answer into a one-string result. This function is taken directly from line 1090 of the "Addition using Numeric Strings" program, except that N is not used since there will never be a final carry. In our subtraction program, concatenation of the individual answers into one result occurs at line 1070.

```
1070 C$=RIGHT$(STR$(C),1)+C$
```

Step 6: Subtraction can generate leading zeros in the answer. We eliminate these leading zeros before printing the answer. The FOR-NEXT loop in lines 1090 to 1120 **checks and eliminates all leading zeros,** using the VAL function and variable L as a counter.

```
1090 FOR I=1 TO LEN(C$)
1100 IF VAL(MID$(C$,I,1))=0 THEN L=L+1
1110 IF VAL(LEFT$(C$,I))<>0 THEN I=LEN(C$)
1120 NEXT I
```

The FOR-NEXT loop, which iterates from 1 to the length of the answer C$, searches for leading zeros or blanks by extracting each digit from C$ and comparing it to zero. It compares digits from left to right. If it identifies a zero or blank, counter variable L is incremented by 1 (statement 1100). As soon as the first non-zero or non-blank character is encountered, loop counter L is set to the length of the string so the program may drop out of the loop immediately.

Once we have determined the number of leading zeros in the answer, we separate the leading zeros from the remainder of the answer C$. At line 1130, the RIGHT$ function takes the LEN(C$) − L rightmost digits and stores them in the answer variable C$.

C$ = [0][0][1][2][3][5][7] LEN(C$) = 7

I	MID$(C$,I,1)		
1	0 0 1 2 3 5 7	= 0	L = 1
2	0 0 1 2 3 5 7	= 0	L = 2
3	0 0 1 2 3 5 7	< > 0	I = LEN(C$)
7			I = 7 drop out of loop

```
1130 C$=RIGHT$(C$,LEN(C$)-L)
```

C$=RIGHT$ ([0][0][1][2][3][5][7],7−2)

C$=RIGHT$ ([0][0][1][2][3][5][7],5)

C$=[1][2][3][5][7]

Step 7: Print the answer string C$. But before we print C$, we check to see if the answer is to be negative by testing variable S at line 1140. If S=1, that means that originally A$<B$, and the final answer is to be negative, so a negative sign is added to C$. If S=0, the answer is positive, so nothing is added. Line 1150 prints C$:

```
1140 IF S=1 THEN C$="-"+C$
1150 PRINT:PRINT"ANSWER=";C$:PRINT
```

The last lines, 1160 through 1180, clear all strings and variables to zero or null, and return the program to the beginning for the next input numbers. The total program is listed below.

```
10 PRINT"[CLR]***SUBTRACTION***":PRINT          Clear screen
20 INPUT A$,B$                                  Input numeric strings
30 BLANK$="                                 "  }
40 X=LEN(A$):Y=LEN(B$)                         } Right justify strings (from lines
50 IF X<Y THEN A$=LEFT$(BLANK$,Y-X)+A$         } 20-60 of the addition program)
60 IF Y<X THEN B$=LEFT$(BLANK$,X-Y)+B$         }
```

```
65 IF VAL(A$)=VAL(B$) THEN C$="0":GOTO 1150
70 IF VAL(A$)>=VAL(B$) GOTO 1000
80 X$=A$:A$=B$:B$=X$                          } If A$<B$, switch strings
90 S=1
1000 REM**SUBTRACTION ROUTINE**
1010 FOR I=LEN(A$) TO 1 STEP-1
1020 A=VAL(MID$(A$,I,1))
1030 A=A+D:D=0                                } Subtraction loop (based on
1040 B=VAL(MID$(B$,I,1))                        lines 1020-1100 of the addition
1050 IF (A-B)<0 THEN D=-1:A=A+10                program)
1060 C=A-B
1070 C$=RIGHT$(STR$(C),1)+C$
1080 NEXT I
1090 FOR I=1 TO LEN(C$)
1100 IF VAL(MID$(C$,I,1))=0 THEN L=L+1        } Truncate leading zeros
1110 IF VAL(LEFT$(C$,I))<>0 THEN I=LEN(C$)      and blanks
1120 NEXT I
1130 C$=RIGHT$(C$,LEN(C$)-L)
1140 IF S=1 THEN C$="-"+C$
1150 PRINT:PRINT"ANSWER=";C$:PRINT            Print answer
1160 C$="":A$="":B$="":X$=""
1165 A=0:B=0:C=0:D=0:S=0:X=0:Y=0              } Clear strings and variables
1170 GOTO20
1180 END
***SUBTRACTION***
?123456789012
??57943572
ANSWER= 123398845440
```

The string subtraction program illustrated above has one problem: it generates a zero result if the subtrahend and minuend have the same number of digits, and in addition are identical in their nine most significant digits. For example, try subtracting 123456789000 from 123456789012. The answer is reported inaccurately as 0. This error results from the statements on line 65. The VAL function computes a 9-digit value for strings A$ and B$. If these two numeric strings are identical in their nine most significant digits, then the equivalence test on line 65 will be true whatever values the two numeric strings may have in lower significant digits. Can you correct this problem by separately testing the upper and lower halves of the numeric strings?

Multiple Integer Subtraction

Recall from the previous discussion of multiple integer addition that the multiple integer method divides a large number into smaller segments, calculates the segments separately, and joins the answers into one string. This method evades the 9-digit length limit.

Multiple Integer Subtraction has these steps:

1. Input the minuend and subtrahend as two positive numeric strings.
2. Determine which string has the larger value.
3. Divide the numbers into high and low parts.
4. Calculate the difference for the low-order and high-order halves.
5. Concatenate the differences into a one-string answer.
6. Truncate leading zeros.
7. Print the answer string.

Step 1: Input the minuend and the subtrahend as two positive numeric strings:

```
10 PRINT"□***MULTIPLE INTEGER SUBTRACTION***":PRINT
20 INPUT A$,B$

RUN

***MULTIPLE INTEGER SUBTRACTION***

?123456789012
??57943572
```

A$, the minuend, and B$, the subtrahend, are entered as strings to avoid the 9-digit length limit.

Like multiple integer addition, A$ and B$ are divided into smaller segments. The maximum input length is arbitrarily set at 16 digits, so that we can divide the largest possible string into equal segments of eight digits each.

Step 2: Determine which input string has the larger value. If A$ is equal to B$ then the program drops down to line 1190 to print a zero answer. If B$ is larger than A$ the difference is negative and extra steps are needed.

If the answer is to be negative, the contents of the two strings are switched to put the larger value in A$ and the smaller value in B$. They are then subtracted, and a negative sign ("−") is concatenated onto the front of the difference (C$) as was demonstrated in line 70 of "Numeric String Subtraction." Line 30 is used here to direct the program past the switching routine if switching is not needed.

```
30 IF VAL(A$)>VAL(B$) THEN 1000
40 X$=A$:A$=B$:B$=X$
50 S=1
```

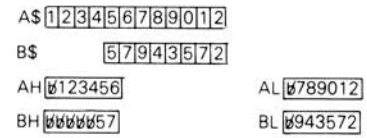
If the value of B$ is larger than the value of A$, the contents of A$ and B$ are switched at lines 40 to 50. This ensures that the smaller number is subtracted from the larger one. A marker is set to indicate that the variables have been switched.

For a detailed explanation of this routine, refer to step 3 of "Numeric String Subtraction."

Step 3: Divide A$ and B$ into two smaller segments, high and low.

```
1000 X=LEN(A$):Y=LEN(B$)
1002 IF X>Y THEN F=X/2:GOTO 1006
1004 F=Y/2
1006 IF F>INT(F) THEN F=INT(F)+1
1010 IF X<=F THEN AH=0:AL=VAL(A$):GOTO 1040
1020 AH=VAL(LEFT$(A$,X-F))
1030 AL=VAL(RIGHT$(A$,F))
1040 IF Y<=F THEN BH=0:BL=VAL(B$):GOTO 1070
1050 BH=VAL(LEFT$(B$,Y-F))
1060 BL=VAL(RIGHT$(B$,F))
```

Statements on lines 1010 and 1040 compare the string lengths with the divider point F. F is determined at lines 1002 and 1006. These lines are identical to lines 1002 and 1006 of the "Multiple Integer Addition" program. If the string is shorter than F, AH (or BH) is assigned a zero value, leaving AL (or BL) with the entire string as its value. If the string is longer than F it must be divided into high and low segments. AH is assigned the leftmost LEN(AH), minus F digits.



Lines 1000 through 1060 are also similar to lines 1000 through 1060 of the "Multiple Integer Addition" program, which divides A$ and B$ into AH, AL, BH, and BL. Refer to step 2 of "Multiple Integer Addition" for further explanation.

Step 4: Calculate differences for the high-order and low-order segments. BL is subtracted from AL, and BH is subtracted from AH:

```
AH[0123456]      AL[0789012]
-BH[0000057]     -BL[0943572]
```

Before the segments are subtracted, the minuend and subtrahend must be compared. If the value of BL is larger than AL the difference is negative. This creates problems because a negative CL cannot be concatenated onto CH:

```
CH[0xxxxxx] ⌒ CL[-xxxxxx] = C[0xxxxxx-xxxxxx]  Incorrect
```

Therefore, **we must borrow from AH to increase the value of AL so that the difference will be positive.** Lines 1070 to 1090 borrow from AH and increase AL before BL is subtracted from AL:

```
1070 IF AL>=BL THEN 1100
1080 AL=AL+10↑F
1090 AH=AH-1
```

If AL is larger than BL we bypass 1080 and 1090 and jump directly to the subtraction. But if BL is larger than AL we must borrow a one million value from AH to increase the value of AL:

```
          -1  ──────────►  +1000000
AH[xxxxx]x       AL[ xxxxxx]
-BH[xxxxxx]      -BL [xxxxxx]
CH[xxxxxx]       CL  [xxxxxx]
```

A ten is added to the leftmost digit of AL. The easiest way to add the ten in the correct position is to raise ten to the Fth power.

```
AL=AL+10↑F
```

In our sample program, AL is smaller than BL, as tested in line 1070.

```
AL[0789012] <BL[0943572]
```

Therefore we must borrow 1000000 (10↑F=10↑6=1000000) from AH to increase the value of AL:

```
1080 AL=AL+10↑F

     AL=AL+10↑6
     AL=AL+1000000
     AL=[789012] + 1000000
     AL=[1789012]
```

After AL is been increased, AH must be decremented by 1, since we borrowed from it.

```
1090 AH=AH-1

     AH=[0123456] -[01]
     AH=[0123455]
```

Once AH, AL, BH, and BL have been set up properly, segments are subtracted. CL$ is the difference between AL and BL, and CH$ is the difference between AH and BH.

Statements on lines 1100 through 1102 compute CL$:

```
1100 CL$=STR$(INT(AL-BL))
     CL$=STR$(ø1789012-ø943572)
     CL$=STR$(ø845340)
     CL$=ø345540
```

Using the MID$ function at line 1101, the leftmost character (a blank representing a positive sign value) is truncated:

```
1101 CL$=MID$(CL$,2,LEN(CL$)-1)
     CL$=MID$(ø8455440,2,6)
     CL$=  8455440
```

At 1102, if the length of CL$ is shorter than F, zeros from ZERO$ are concatenated onto the front of CL$. An assignment statement assigns a string of 0s to variable ZERO$ on line 15. In this case, the length of CL$ is equal to F, therefore no leading zeros are needed.

```
15 ZERO$="0000000000000000"
1102 CL$=LEFT$(ZERO$,F-LEN(CL$))+CL$
     CL$=LEFT$ (ZERO$,6-6)+CL$
     CL$=LEFT$ (ZERO$,0)+CL$
```

At line 1110, CH$ is assigned the string integer value of AH−BH:

```
1110 CH$=STR$(INT(AH-BH))
     CH$=STR$(ø123455-ø57)
     CH$=STR$(ø123398)
     CH$=ø123395
```

Using the MID$ function, the leftmost blank character is truncated:

```
1111 CH$=MID$(CH$,2,LEN(CH$)-1)
     CH$=MID$(ø123398,2,6)
     CH$=123398
```

The subtraction routine looks like this:

```
1070 IF AL>=BL GOTO 1100
     789012>=943572 ──────→ False statement
     Program continues at next line
1080 AL=AL+10↑F
     AL=789012+1000000
     AL=1789012
1090 AH=AH-1
     AH=123456-1
     AHø123455
1100 CL$=STR$(INT(AL-BL))
     CL$=STR$(ø1789012-ø943572)
     CL$=STR$(ø845540)
     CL$=845540
1101 CL$=MID$(CL$,2,LEN(CL$)-1)
     CL$=MID$(ø845540,2,7-1)
     CL$=MID$(ø845540,2,6)
     CL$=845540
```

```
1102 CL$=LEFT$(ZERO$,F-LEN(CL$))+CL$
     CL$=LEFT$(ZERO$,6-6)+CL$
     CL$=LEFT$(ZERO$,0)+[8][4][5][5][4][0]
     CL$=[8][4][5][5][4][0]
1110 CH$=STR$(INT(AH-BH))
     CH$=STR$(ø123455-ø57)
     CH$=STR$(ø123398)
     CH$=[ø][1][2][3][3][9][8]
1111 CH$=MID$(CH$,2,LEN(CH$)-1)
     CH$=MID$([ø][1][2][3][3][9][8],2,7-1)
     CH$=MID$([ø][1][2][3][3][9][8],2,6)
     CH$=[1][2][3][3][9][8]
```

Step 5: Concatenate the answer strings, CH$ and CL$, together by numeric string concatenation. They are concatenated in statement 1120:

```
1120 C$=CH$+CL$
     C$=CH$[    ]+ CL$[    ]
     C$= [          ]
```

Only the rightmost "F" numbers from CL$ are concatenated onto CH$ to avoid concatenating any leading blanks in CL$ (see the "Subtraction using Numeric Strings" section for further discussion).

Step 6: Truncate leading zeros in C$ before C$ is printed. Leading zeros are subtracted in the same way for Multiple Integer Subtraction as for Numeric String Subtraction (see step 5 of "Subtraction using Numeric Strings"). Lines 1130 through 1170 truncate leading zeros just prior to printing C$:

```
1130 FOR I=1 TO LEN(C$)
1140 IF VAL(MID$(C$,I,1))=0 THEN L=L+1
1150 IF VAL(LEFT$(C$,I))<>0 THEN I=LEN(C$)
1160 NEXT I
1170 C$=RIGHT$(C$,LEN(C$)-L)
1180 IF S=1 THEN C$="-"+C$
```

If A$ and B$ had been switched, S would have been set to 1, signaling a negative answer, and thus a negative sign would be concatenated onto the front of C$ at 1180.

Step 7: Print the answer and clear out variable strings before allowing another problem to be input.

```
1190 PRINT:PRINT"ANSWER= ";C$:PRINT
1200 A$="":B$="":C$="":CH$="":CL$=""
1205 AH=0:AL=0:BH=0:BL=0:F=0:S=0:X=0:Y=0
1210 GOTO 20
1220 END
```

The finished program appears as follows:

```
10 PRINT"□***MULTIPLE INTEGER SUBTRACTION***":PRINT
15 ZERO$="0000000000000000"
20 INPUT A$,B$
25 IF VAL(A$)=VAL(B$) THEN C$="0":GOTO 1190
30 IF VAL(A$)>VAL(B$) GOTO 1000
40 X$=A$:A$=B$:B$=X$
50 S=1
1000 X=LEN(A$):Y=LEN(B$)
1002 IF X>Y THEN F=X/2:GOTO 1006
1004 F=Y/2
```

```
1006 IF F>INT(F) THEN F=INT(F)+1
1010 IF X<=F THEN AH=0:AL=VAL(A$):GOTO 1040
1020 AH=VAL(LEFT$(A$,X-F))
1030 AL=VAL(RIGHT$(A$,F))
1040 IF Y<=F THEN B=0:BL=VAL(B$):GOTO 1070
1050 BH=VAL(LEFT$(B$,Y-F))
1060 BL=VAL(RIGHT$(B$,F))
1070 IF AL>=BL GOTO 1100
1080 AL=AL+10↑F
1090 AH=AH-1
1100 CL$=STR$(INT(AL-BL))
1101 CL$=MID$(CL$,2,LEN(CL$)-1)
1102 CL$=LEFT$(ZERO$,F-LEN(CL$))+CL$
1110 CH$=STR$(INT(AH-BH))
1111 CH$=MID$(CH$,2,LEN(CH$)-1)
1120 C$=CH$+CL$
1130 FOR I=1 TO LEN(C$)
1140 IF VAL(MID$(C$,I,1))=0 THEN L=L+1
1150 IF VAL(LEFT$(C$,I))<>0 THEN I=LEN(C$)
1160 NEXT I
1170 C$=RIGHT$(C$,LEN(C$)-L)
1180 IF S=1 THEN C$="-"+C$
1190 PRINT:PRINT"ANSWER= ";C$:PRINT
1200 A$="":B$="":C$="":CH$="":CL$=""
1205 AH=0:AL=0:BH=0:BL=0:F=0:S=0:X=0:Y=0
1210 GOTO 20
1220 END

***MULTIPLE INTEGER SUBTRACTION***

?123456789012
??57943572

ANSWER= 123398845440

?1234567890123456
??57943572

ANSWER= 1234567832179884

?9999999999999999
??1234567890

ANSWER= 9999998765432109
```

You now know two methods of subtraction. The first method used numeric strings. The second uses multiple integer math. By comparing their outputs, you can see that both methods work equally well at getting around the 9-digit length limit.

MULTIPLICATION

A 9-digit length limit may be easily exceeded by multiplication because a product may be very large, even when the multiplier and multiplicand are small. This numeric length limit prohibits products longer than nine digits from being displayed without exponential notation. You can get around this limitation by writing a program that displays products with more than nine digits of precision. Displaying products exceeding nine digits without exponential notation is most easily done using Multiple Integer Multiplication. **The following program and discussion will enable you to display products up to 16 digits in length without exponential notation.**

Multiple Integer Multiplication

Using virtually the same steps as Multiple Integer Addition and Subtraction, Multiple Integer Multiplication separates the multiplicand and multiplier into smaller segments, multiplies all segments, and adds the multiple products together into one final product, which can have from one to 16 digits.

The steps for Multiple Integer Multiplication are as follows:

1. Input the multiplicand and the multiplier as two positive numeric strings.
2. Divide the strings into high and low segments.
3. Multiply the corresponding segments.
4. Add the segment products to create one product string. Truncate any leading zeros.
5. Print the product string.

Step 1: Input the multiplicand and the multiplier as two positive numeric strings, where A$ is the multiplicand and B$ is the multiplier. As with the other math programs, the numbers are input as strings to avoid the 9-digit length limit.

This program limits the length of the product to 16 digits. Since the maximum product length equals the sum of the lengths of the multiplicand and multiplier, *the sum of the lengths of the input numbers cannot exceed 16.* Changing the program to accept larger numbers requires several alterations which will not be discussed; you should be able to make such changes yourself. For this program:

(length of A$)+(length of B$) $\leq$ 16

Examples: 12 + 4 $\leq$ 16
2 + 3 $\leq$ 16
8 + 8 $\leq$ 16

The example program will multiply two input numbers with equal lengths of eight digits: 99999999 and 99999999, to give us a 16-digit product.

```
  99999999←        8 digits
x 99999999←   +    8 digits
----------------------------
9999999800000001←  16 digits
```

Input the multiplier and multiplicand as two positive numeric strings, A$ and B$:

```
10 PRINT "***MULTIPLE INTEGER MULTIPLICATION***":PRINT
20 INPUT A$,B$
RUN
***MULTIPLE INTEGER MULTIPLICATION***
?99999999
??99999999
```

Step 2: Separate both input strings into two segments: high (H) for the leftmost digits and low (L) for the rightmost digits. The dividing point, variable F, specifies where to divide A$ and B$ into segments. The value of F is set at lines 1002 and 1006 (for explanation refer to "Multiple Integer Addition").

```
1000 X=LEN(A$):Y=LEN(B$)
     X=8          Y=8
1002 IF X>Y THEN F=X/2:GOTO 1008
1004 F=Y/2
     F=8/2
     F=4
1006 IF F>INT(F)THEN F=INT(F)+1
```

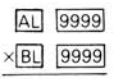
Once F is set, the program divides the numbers into high and low segments. This routine was presented in the "Multiple Integer Addition" program. Lines 1010 through 1060 divide the two strings into high and low segments.

```
1010 IF X<=F THEN AH=0:AL=VAL(A$):GOTO 1040
1020 AH=VAL(LEFT$(A$,X-F))
1030 AL=VAL(RIGHT$(A$,F))
1040 IF Y<=F THEN BH=0:BL=VAL(B$):GOTO 1070
1050 BH=VAL(LEFT$(B$,Y-F))
1060 BL=VAL(RIGHT$(B$,F))
```

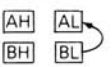
The routine above divides A$ into AH and AL (four digits) and B$ into BH and BL (four digits):

Step 3: Multiply AH, AL, BH, and BL into four product strings: P1$, P2$, P3$, and P4$. The rules of algebraic multiplication multiply each variable as if it were a single number. A$ and B$ are multiplied as follows:

Think of A$ and B$ as two sets of 4-digit numbers (H and L) joined in the middle, and not as eight individual digits: A$ is not eight 9s, but two sets of four 9s each. Thus AL and BL are multiplied as:



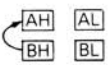
Multiplying A$ and B$ is a four-step process. To begin, multiply BL by AL:



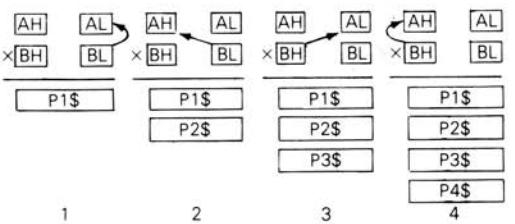
and then multiply BL by AH:

Next, move over to BH and multiply BH by AL:

and finally multiply BH by AH:



Here is the four-step process:



Let's look step by step at how the multiplication works, using the values of AH, AL, BH, and BL from our example:

AH 9999 AL 9999
BH 9999 BL 9999

The first multiplication is BL times AL:

```
 AH  9999
 BH  9999
-----------
      89991
     89991
    89991
   89991
-----------
   99980001
```

The second multiplication is BL times AH, as shown in the diagram below:

```
AH 9999    AL
 x  BH  BL 9999
-----------------
        99980001
999800010000
```

Notice that P2 is not directly beneath P1, but four spaces to the left (recall the rules for lining up the products of 2-digit multiplication problems). To continue in the same manner, the third multiplication should be as follows:

```
  AH   AL 9999
BH 9999   BL
-------------
  999800010000
```

The fourth and final multiplication should be as follows:

```
AH 9999   AL
BH 9999   BL
-------------
9998000100000000
```

Remember that only the values of the four segments are multiplied; this means that the actual multiplication done by AL × BH, etc. yields the same number, 99980001, for all four products. In the program the products are aligned by converting the products into strings and concatenating the necessary number of zeros onto the end of the strings. This aligns the strings correctly. Statements on lines 1070 through 1100 perform this alignment:

```
1070 P1$=STR$(BL*AL)
1080 P2$=STR$(BL*AH)+F$
1090 P3$=STR$(BH*AL)+F$
1100 P4$=STR$(BH*AH)+F$+F$
```

Without alignment the answers would be computed incorrectly as follows:

```
P1  99980001
P2  99980001   Incorrect
P3  99980001
P4  99980001
-----------
            
```

instead of:

```
          99980001
   99980001 0000
   99980001 0000      Correct
99980001 00000000
-----------------
```

The number of zeros to be concatenated onto the end of the product strings is assigned to F$. F$ contains F zeros. F equals the number of digits in each half of the multiplier and multiplicand.

```
40 ZERO$="0000000000000000"
1008 F$=LEFT$(ZERO$,F)
     F$=LEFT$(ZERO$,4)
     F$=0000
     F$="0000"
```

When P1$, P2$, P3$, and P4$ are computed (lines 1070 through 1100), the correct number of zeros are simultaneously concatenated to the end of the string to align the products correctly. The products are now aligned and ready to be added:

```
   AH 9999    AL 9999
 x BH 9999    BL 9999
-------------------------
          99980001   P1
     999800010000    P2
     999800010000    P3
9998000100000000    P4
          F$  F$
```

At the end of step 3, the program looks like this:

```
20 INPUT A$,B$                                    Input values for A$, B$
30 IF VAL(A$)=0 OR VAL(B$)=0 THEN                 If multiplicand or
   C$="0":GOTO 1190                                 multiplier = 0 then
40 ZERO$="0000000000000000"                         answer (C$) = 0
1000 X=LEN(A$):Y=LEN(B$)
1002 IF X>Y THEN F=X/2:GOTO 1006                  Set divider point F
1004 F=Y/2
1006 IF F>INT(F)THEN F=INT(F)+1
1008 F$=LEFT$(ZERO$,F)
1010 IF X<=F THEN AH=0:AL=VAL(A$):GOTO 1040       Divide A$ and B$
1020 AH=VAL(LEFT$(A$,X-F))                          into parts
1030 AL=VAL(RIGHT$(A$,F))                           high and low
1040 IF Y<=F THEN BH=0:BL=VAL(B$):GOTO 1070
1050 BH=VAL(LEFT$(B$,Y-F))
1060 BL=VAL(RIGHT$(B$,F))
1070 P1$=STR$(BL*AL)                              Multiply A$ and B$
1080 P2$=STR$(BL*AH)+F$                             and align products
1090 P3$=STR$(BH*AL)+F$
1100 P4$=STR$(BH*AH)+F$+F$
```

Step 4: Add the four products together. This is the most complicated part of the "Multiple Integer Multiplication" program because parameters are passed back and forth from the main program to an addition subroutine. We will use a portion of the

"Addition using Numeric Strings" program as a subroutine to add the products together. Below is the portion of the addition program we will be using as a subroutine:

```
2000 REM**ADD PRODUCTS**
2010 BLANK$="                    "
2020 X=LEN(A$):Y=LEN(B$)
2030 IF X<Y THEN A$=LEFT$(BLANK$,Y-X)+A$
2040 IF X>Y THEN B$=LEFT$(BLANK$,X-Y)+B$
2050 D=0:N=1:C$=""
2060 FOR I=LEN(A$) TO 1 STEP-1
2070 A=VAL(MID$(A$,I,1))
2080 A=A+D:D=0
2090 B=VAL(MID$(B$,I,1))
2100 C=A+B
2110 IF C>=10 THEN D=1
2120 IF D=1 AND I=1 THEN N=2
2130 C$=RIGHT$(STR$(C),N)+C$
2140 NEXT I
```

At line 1110 the contents of P1$ and P2$ are passed to the parameters A$ and B$, which are used in the addition subroutine (lines 2000 and 2140).

```
1110 A$=P1$:B$=P2$
```

A$ 9998001

B$ 99980001 0000

Notice that the contents of A$ and B$ are not the same as those input at line 20. The same variable names are used to allow program compatibility between all four math programs. Only two parameters are passed at a time because the addition subroutine adds only two numbers at a time.

Once the values for P1$ and P2$ are passed to A$ and B$ the addition subroutine is called:

```
1120 GOSUB 2000
```

A$ and B$ are right-justified and equated in length for addition by adding blanks from BLANK$ to the shorter string (if there is one) in lines 2010 to 2040:

```
2010 BLANK$="                    "
2020 X=LEN(A$):Y=LEN(B$)
2030 IF X<Y THEN A$=LEFT$(BLANK$,Y-X)+A$
2040 IF X>Y THEN B$=LEFT$(BLANK$,X-Y)+B$
```

Statements 2050 to 2140 add the corresponding digits of A$ and B$ and convert the sum C into the numeric string C$. (A full explanation of this process is given in the "Addition using Numeric Strings" section.)

```
2050 D=0:N=1:C$=""
2060 FOR I=LEN(A$) TO 1 STEP-1
2070 A=VAL(MID$(A$,I,1))
2080 A=A+D:D=0
2090 B=VAL(MID$(B$,I,1))
2100 C=A+B
2110 IF C>=10 THEN D=1
2120 IF D=1 AND I=1 THEN N=2
2130 C$=RIGHT$(STR$(C),N)+C$
2140 NEXT I
```

The sum, C$, is passed through a FOR-NEXT loop to truncate any leading blanks or zeros at lines 3000 to 3060. This truncation routine is from "Subtraction using Numeric Strings."

```
3000 REM***TRUNCATE LEAD ZEROS***
3001 L=0
3010 FOR I=1 TO LEN(C$)
3020 IF VAL(MID$(C$,I,1))=0 THEN L=L+1
3030 IF VAL(LEFT$(C$,I))<>0 THEN I=LEN(C$)
3040 NEXT I
3050 C$=RIGHT$(C$,LEN(C$)-L)
3060 RETURN
```

C$, the sum of P1$ and P2$, is returned to the main program and converted to M1$:

```
1130 M1$=C$
```

The contents of C$ must be transferred to M1$ because C$ must be cleared before the addition subroutine is called again at line 1150 to add P3$ and P4$.

To add P3$ and P4$ together, the values of P3$ and P4$ are passed to the parameters A$ and B$ before calling the addition subroutine 2000:

```
1132 A$=P3$:B$=P4$:GOSUB 2000
```

A$ [9][9][9][8][0][0][0][0][0][0][0]

B$ [9][9][9][8][0][0][0][1][0][0][0][0][0][0][0]

The addition subroutine adds the corresponding digits of P3$ and P4$, truncates any leading zeros, and returns sum C$ to the main program, where C$ is converted to M2$:

```
1135 M2$=C$
```

The addition subroutine is called a third time to add M1$ and M2$ together to get the final answer, C$.

```
1140 A$=M1$:B$=M2$
```

A$ [9][9][9][8][9][9][9][9][0][0][0][1]

B$ [9][9][9][8][9][9][9][9][0][0][0][1][0][0][0][0]

```
1150 GOSUB 2000
```

Step 5: After the third return from the addition subroutine, C$ equals the sum of all four products. Step 5 **prints the answer.** The GOTO 20 allows another multiplication problem to be solved.

```
1190 PRINT:PRINT"ANSWER=";C$:PRINT:GOTO 20
1200 END
```

The flow of the program looks like this:

```
Step 1   10  PRINT" ***MULTIPLE INTEGER MULTIPLICATION***"
         20
         30    Input multiplier and multiplicand,
         40    initialize variables
         50

Step 2   1000
         1010
         1020
         1030  Calculate F, divide the multiplier, multiplicand
         1040  into high and low segments
         1050
         1060

Step 3   1070
         1080  Multiply segments into four products
         1090  into four products P1$, P2$, P3$, P4$
         1100

Step 4   1110  Pass P1$ + P2$ to parameters A$, B$
         1120  GOSUB 2000 ------------> 2000-2140 addition subroutine;
                                        add P1$ + P2$ → C$
                <------------------------ 3000-3060 truncate leading zeros
         1130  Pass contents of C$ →M1$
         1140  Pass P3$ + P4$ to parameters A$, B$
         1150  GOSUB 2000 ------------> 2000-2140 addition subroutine;
                                        add P3$ + P4$ → C$
                <------------------------ 3000-3060 truncate leading zeros
         1160  Pass contents of C$ →M2$
         1170  Pass M1$ + M2$ to parameters A$, B$
         1180  GOSUB 2000 ------------> 2000-2140 addition subroutine;
                                        add M1$ + M2$ → C$
                <------------------------ 3000-3060 truncate leading zeros

Step 5   1190  Prints C$
         1200  END
```

Here is the multiplication program listing and sample run:

```
10 PRINT"□***MULTIPLE INTEGER MULTIPLICATION***":PRINT
20 INPUT A$,B$
30 IF VAL(A$)=0 OR VAL(B$)=0 THEN C$="0":GOTO 1190
40 ZERO$="0000000000000000"
1000 X=LEN(A$):Y=LEN(B$)
1002 IF X>Y THEN F=X/2:GOTO 1008
1004 F=Y/2
1006 IF F>INT(F)THEN F=INT(F)+1
1008 F$=LEFT$(ZERO$,F)
1010 IF X<=F THEN AH=0:AL=VAL(A$):GOTO 1040
1020 AH=VAL(LEFT$(A$,X-F))
1030 AL=VAL(RIGHT$(A$,F))
1040 IF Y<=F THEN BH=0:BL=VAL(B$):GOTO 1070
1050 BH=VAL(LEFT$(B$,Y-F))
1060 BL=VAL(RIGHT$(B$,F))
1070 P1$=STR$(BL*AL)
1080 P2$=STR$(BL*AH)+F$
1090 P3$=STR$(BH*AL)+F$
1100 P4$=STR$(BH*AH)+F$+F$
1110 A$=P1$:B$=P2$
1120 GOSUB 2000
1130 M1$=C$
1132 A$=P3$:B$=P4$:GOSUB 2000
1135 M2$=C$
1140 A$=M1$:B$=M2$
1150 GOSUB 2000
1190 PRINT:PRINT"ANSWER=";C$:PRINT:GOTO 20
1200 END
2000 REM**ADD PRODUCTS**
2010 BLANK$="                    "
2020 X=LEN(A$):Y=LEN(B$)
2030 IF X<Y THEN A$=LEFT$(BLANK$,Y-X)+A$
2040 IF X>Y THEN B$=LEFT$(BLANK$,X-Y)+B$
2050 D=0:N=1:C$=""
2060 FOR I=LEN(A$) TO 1 STEP-1
2070 A=VAL(MID$(A$,I,1))
2080 A=A+D:D=0
2090 B=VAL(MID$(B$,I,1))
2100 C=A+B
2110 IF C>=10 THEN D=1
2120 IF D=1 AND I=1 THEN N=2
2130 C$=RIGHT$(STR$(C),N)+C$
2140 NEXT I
3000 REM***TRUNCATE LEAD ZEROS***
3001 L=0
3010 FOR I=1 TO LEN(C$)
3020 IF VAL(MID$(C$,I,1))=0 THEN L=L+1
3030 IF VAL(LEFT$(C$,I))<>0 THEN I=LEN(C$)
3040 NEXT I
3050 C$=RIGHT$(C$,LEN(C$)-L)
3060 RETURN


*** MULTIPLE INTEGER MULTIPLICATION***

?999999999
??999999999

ANSWER= 999999980000000l
```

GRAPHICS

Computer graphics is a unique subject. Whole books are devoted to this subject. Of necessity, the discussion that follows is brief.

The standard graphic character set includes 64 graphic symbols. Select graphics by issuing a POKE 59468,12 if you are using the alternate character set, which has very few graphic characters. **If you have a CBM 8000 computer, select graphics using the Graphic editing function,** as follows:

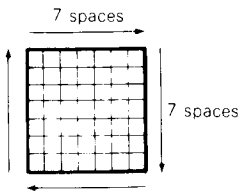
```
100 print chr$(142):rem select graphics
```

The graphic characters are all located in the upper-case positions on the keys, so they must be entered in shifted mode.

Many graphic characters are referenced and illustrated on the following pages. Refer to Table 1-1 or Appendix A for easy reference to graphic character keys, names, and symbols.

GRAPHICS IN IMMEDIATE MODE

Sketching in immediate mode requires no line numbers, no PRINT statements, and no quotation marks. In immediate mode the cursor may be moved freely up, down, right, or left to any spot on the screen without pressing the RETURN key after each directional change. Below is an example of a square drawn in immediate mode. Starting with the cursor in home position, the square was drawn left to right, top to bottom, right to left, and bottom to top, in one continuous movement. No line numbers, program statements, or carriage returns were needed.



We will use the square shown above as the basic graphic design to illustrate elementary graphics. Though simple in its design, sketching this square uses all CBM computer graphic drawing techniques.

Draw a Square

There are nine steps to drawing a 7 × 7 square. They are:

Step 1: HOME the cursor. The top left corner of the HOME position space becomes the top left corner of the square (Figure 5-3a).

Step 2: Type the upper left corner of the square. This is done by using the TOP LEFT CORNER □ (Figure 5-3b).

Step 3: Draw the top line of the square. Because we will use a CORNER key for the top right corner, type five TOP LINE HORIZONTAL □ characters in this step (Figure 5-3c).

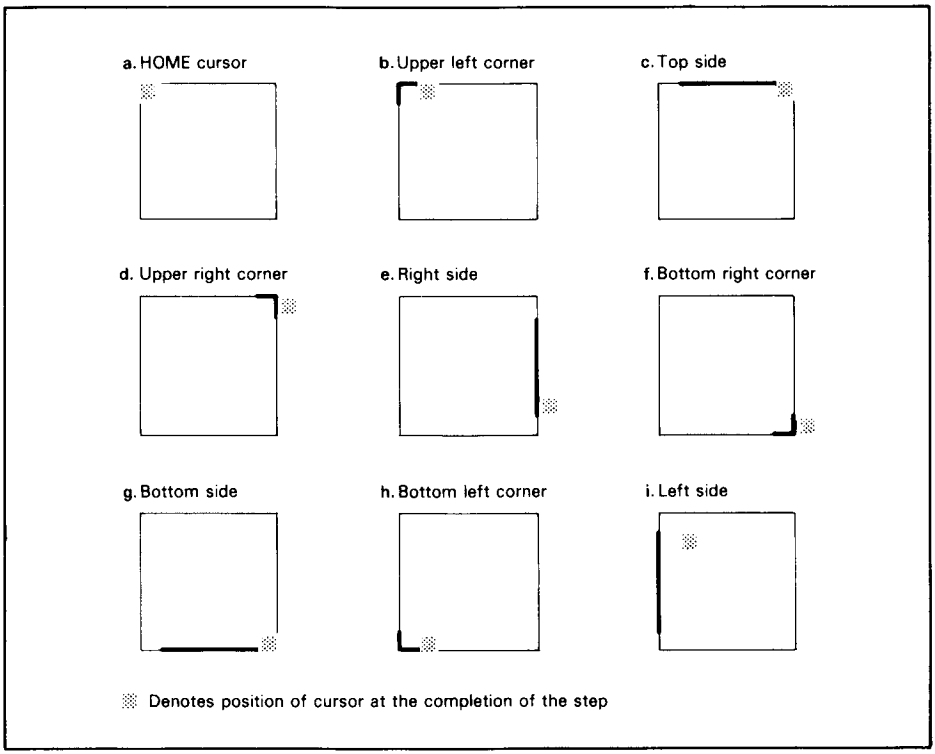


Figure 5-3. Draw the Square

Step 4: Type the upper right corner of the square using the TOP RIGHT CORNER character ❐ (Figure 5-3d).

Step 5: Draw the vertical right side of the square. To allow space for the corner key, type five RIGHT LINE VERTICAL ❐ .

We all know what this part of the square should look like, but does your screen look like this instead?

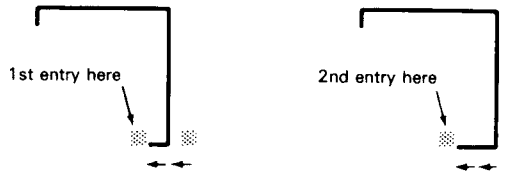
If so, this happened because the cursor is automatically moved one space to the right after any character is displayed. To enter characters vertically, the cursor must be repositioned both vertically and horizontally to compensate for the automatic cursor movement to the right.

To print the vertical line of the square, then, repeat the sequence of CURSOR DOWN, CURSOR LEFT, and RIGHT LINE VERTICAL. Do this five times, and you should have printed the right side of the square (Figure 5-3e).

Step 6: Type the bottom right corner of the square using the BOTTOM RIGHT CORNER character ❐ . Before you type this, look to see where your cursor is; if you haven't already done so, use CURSOR DOWN and CURSOR LEFT to position the cursor at the corner of the square; then press the corner key (Figure 5-3f).

Step 7: Draw the bottom line. Because we are using CORNER keys, we need just five BOTTOM LINE HORIZONTAL characters ❐ (Figure 5-3g).

One method is to enter the line from right to left. After each character entry on the bottom line, two CURSOR LEFT movements will be needed to correctly position the cursor for the next entry.



A second, and possibly more natural, method of drawing the bottom line is from left to right. To do this, position the cursor to the leftmost space of the bottom line (one space to the right of the left edge of the screen); this can be done using six CURSOR LEFTs. You can then easily enter five BOTTOM LINE HORIZONTALs to create the bottom line of the square.

Step 8: Type the bottom left corner. Depending on which method you used to enter the bottom line, you will need to use CURSOR LEFT two times (method 1) or six times (method 2) to position the cursor at the bottom left corner, then use the BOTTOM LEFT CORNER character □ to complete this step (Figure 5-3h).

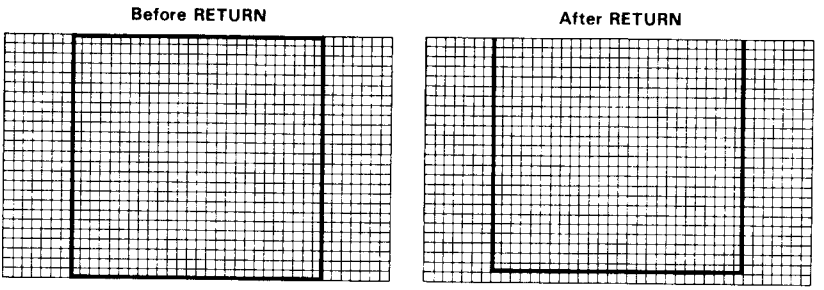
Step 9: Complete the square by drawing the left vertical side. You should be able to type five LEFT LINE VERTICAL characters □ to complete the square (Figure 5-3i). You will need to position the cursor before each entry, using CURSOR LEFT and CURSOR UP.

PROGRAMMING GRAPHICS

Any graphics sketched directly onto the screen will be lost when you execute a NEW statement or turn the power off, unless you first convert the graphics into a program. **You can convert any design sketched onto the screen into a program simply by making each line on the screen a string which is to be printed as part of a program.**

After you have sketched the square, move the cursor to the HOME position. *Do not* press the CLEAR or RETURN key. If you press CLEAR you will lose your picture forever. If you press RETURN, "READY" will be written through the middle of the square as shown below:

Or, if you had made your square so large that the horizontal lines of the square were printed on the top and bottom rows of the screen, and the cursor was positioned on the bottom line, a RETURN would cause the display to scroll up one line in order to write the READY message on the next line, losing the top of the picture.



For this reason, pictures larger than 39 characters wide or 24 characters long should never be drawn in immediate mode.

Once the cursor is homed, the next step is to move each line of the picture to the right in order to insert line numbers, question marks (shorthand for PRINT) and quotes. This converts each line from immediate mode to program mode so it may be saved on a cassette tape or diskette.

When the cursor has been homed, it should be at the upper left corner of the square (Figure 5-4a). Press INSERT five times so that the top line of the square is shifted five spaces to the right (Figure 5-4b). Now there is enough room to type a line number (100), ?, and opening string quotes (Figure 5-4c). Then press RETURN (Figure 5-4d). The top line of the square is now a programmed statement. Continue doing this for each line, incrementing each line number by 100 until the entire square has been converted into program statements (Figure 5-4e, f).

Be sure to number the lines in sequential order to avoid distorting the picture. Also, you do not need to move the cursor past the graphics to insert a second set of quotation marks at the end of each line. After the first set of quotes is typed, merely press RETURN. Your final program listing should appear as follows:

```
100 PRINT"┌──────┐
200 PRINT"│      │
300 PRINT"│      │
400 PRINT"│      │
500 PRINT"│      │
600 PRINT"│      │
700 PRINT"└──────┘
```

Instead of creating graphics in immediate mode and converting them to a program, you can skip immediate mode completely. To draw the picture in program mode, each line of the picture is entered as part of a PRINT statement.

```
100 ?"┌──────┐
200 ?"│      │
300 ?"│      │
```

The space directly to the right of the quotation marks becomes column number 1 on th . screen. If you do not program with this in mind, your picture may end up shifted to the left-hand side of the screen.

If you PRINT a string that has exactly 40 characters, you must include a second set of quotes, and a semicolon at the end of the line. If you do not include the semicolon, an extra line will be displayed since the cursor automatically positions to the next line after a display in column 40.

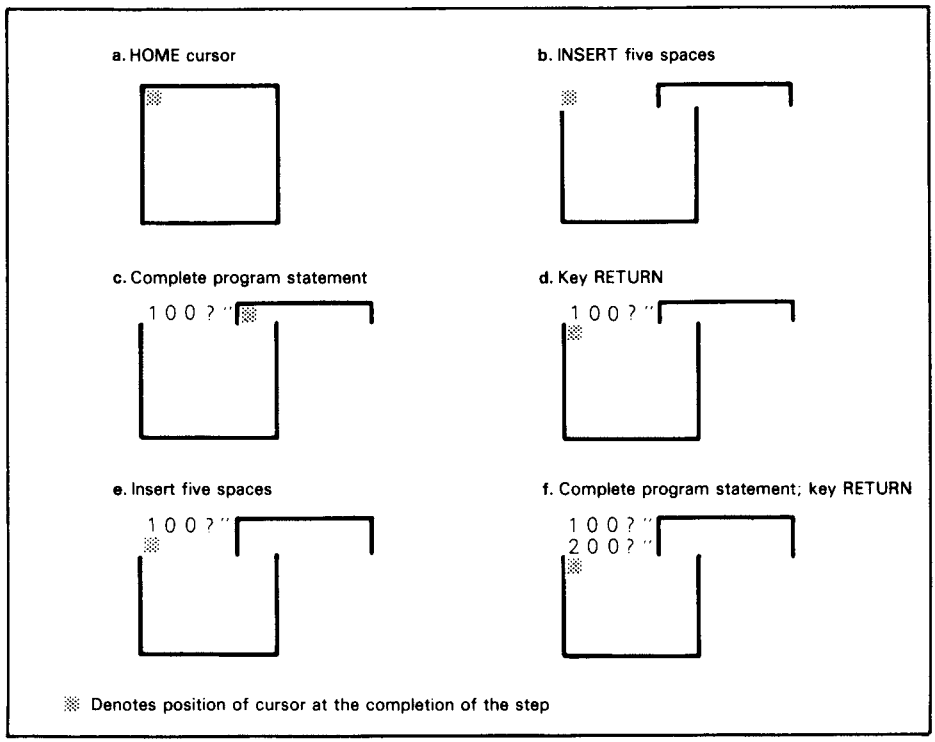


Figure 5-4. Make Program Statement from Graphics

A hint before moving to the next aspect of graphics: it is advisable to draw your picture or diagram on a piece of paper before drawing it on the screen. Map out on a piece of graph paper an area 40 squares wide by 25 squares long, using one square on paper for each space on the screen. Be sure to include space for the line numbers if you are going to convert the picture to program mode. Once everything is ready, type the program from the paper onto the screen.

ANIMATION

Any graph, number, design, word or picture may be programmed to move sideways, up, down, or diagonally, flash on and off, or display more slowly. These changes may be programmed in almost any combination.

To demonstrate animation, we will begin by animating the small square programmed in the previous section. **Instead of seeing the square appear instantaneously on the screen, animation will allow a viewer to watch each element of the square slowly appear on the screen.**

The program to animate the square looks very different from the previous program because the line segments are programmed as BASIC statements, rather than as picture segments. There is no large square within quotation marks; the square is broken down into individual graphic characters.

Time Delay

The animation program slowly moves the cursor so that the square appears to be drawn on the screen. The display begins at the top left corner of the screen and proceeds clockwise, as follows:

The first step, as always, is to clear the screen. This also puts the cursor in the home position.

```
5 PRINT"■";
```

The second step is to type the left corner. However, do not draw the whole top line as you did in the previous program, just the corner.

```
10 PRINT"┌";
```

In order to see each element of the square being displayed, it is necessary to slow down statement execution. This can be done by using a time delay loop. This statement represents one way of creating a time delay:

```
100 FOR J=1 TO 100:NEXT J:RETURN
```

The FOR-NEXT loop increases the time that separates display of adjacent characters. It forces the computer to count from 1 to 10 each time the statement is executed as a subroutine. The TO index for J can be increased or decreased to lengthen or shorten the delay. The larger the TO index, the longer the time separating the display of each character.

For our animation program, then, we must include this time delay loop after displaying each element. Since the programmed time delay loop remains the same for each element, we call it as a subroutine. Therefore, after displaying the upper left corner of the square, call the time delay loop as subroutine 100.

Programming Character Placement

The third phase is to print the top line of the square. Instead of programming PRINT " ─ " we will use a FOR-NEXT loop:

```
15 FOR I=1 TO 5:PRINT"─";:GOSUB 100:NEXT I
```

Statement 15 uses a FOR-NEXT loop so that the subroutine time delay can be called between each printing of "─". If the computer is to sketch the square slowly, the time delay must be called after *each* character is displayed. It would be useless to program:

```
15 PRINT"─────";:GOSUB 100 ← Incorrect
```

because the whole line would be printed instantaneously without any time delay.

To complete the top line, type the upper right corner. Again, include the time delay subroutine call:

```
20 PRINT"┐";:GOSUB 100
```

So far, the program looks like this:

```
5 PRINT"■";
10 PRINT"┌";:GOSUB 100
15 FOR I=1 TO 5:PRINT"─";:GOSUB 100:NEXT I
20 PRINT"┐";:GOSUB 100
30 END
100 FOR I=1 TO 100:NEXT J:RETURN
```

Run the program. You should see the following display grow progressively, from ┌ to ┌──────┐ .

Hopefully, this is what you saw. If not, go back and find out what went wrong. Did you forget the semicolons after each PRINT statement?

End all PRINT statements in this program with a semicolon (;). The semicolon concatenates graphic strings together when printed. This allows the " ┌ " and the top line "──────" to be concatenated together on the same line. Without the semicolons, the CBM computer performs a carriage return after each statement, and the top line will look like this:

```
┌
─
─
─
─
─
┐
▒
```

The other three sides are drawn using a similar sequence. Line 20 begins the next sequence, to create the right side vertical line. Note the use of cursor control inside the FOR-NEXT loops to compensate for the automatic right cursor movement.

Here are the PRINT statements that must appear within FOR-NEXT loops to generate the right side, bottom and left side of the square:

PRINT" ■■"	right side	PRINT<RIGHT LINE VERT ><CURSOR L > <CURSOR DOWN>
PRINT"_■■"	bottom	PRINT<BOTTOM LINE HORIZ ><CURSOR L > <CURSOR L >
PRINT"▌■"	left side	PRINT<LEFT LINE VERT ><CURSOR L > <CURSOR UP>

The complete program listing looks like this:

```
5 PRINT"■";
10 PRINT"┌"; GOSUB 100
15 FOR I=1TO 5 PRINT"─"; GOSUB 100 NEXT I
20 PRINT"┐"; GOSUB 100
25 FOR I=1TO 5 PRINT" ■■"; GOSUB 100 NEXT I
30 PRINT"┘"; GOSUB 100
35 FOR I=1TO 5 PRINT"_■■"; GOSUB 100 NEXT I
40 PRINT"└"; GOSUB 100
45 FOR I=1TO 5 PRINT"▌■"; GOSUB 100 NEXT I
50 END
100 FOR J=1 TO 10 NEXT J RETURN
```

Now try a trial run. Does your square look like this?

```
┌──────┐ ▌
READY.  ▌
    ▌   ▌
    ▌   ▌
    ▌   ▌
  └ ▌ ____
```

If this design appears instead of a perfect square, some of the cursor controls were left out. The computer did exactly what it was programmed to do, so where is the problem? Take a closer look at the program. We included cursor controls within the FOR-NEXT loops for all four sides of the square. Now look at the screen. The problem is not with the sides; therefore the problem must be in the corners. Look at statements 20, 30, and 40.

We forgot the cursor controls after each corner position. Make the proper changes, and the program should look like this:

```
5 PRINT"□";
10 PRINT"┌";:GOSUB 100
15 FOR I=1 TO 5:PRINT"─";:GOSUB 100:NEXT I
20 PRINT"┐";:GOSUB 100
25 FOR I=1 TO 5:PRINT" │";:GOSUB 100:NEXT I
30 PRINT"_┘";:GOSUB 100
35 FOR I=1 TO 5:PRINT"_─";:GOSUB 100:NEXT I
40 PRINT"└┌";:GOSUB 100
45 FOR I=1 TO 5:PRINT"│ ";:GOSUB 100:NEXT I
50 END
100 FOR I=1 TO 100:NEXT J:RETURN
```

Now try another trial run. Your picture should look like this:

```
┌──────┐
READY. │
▒      │
│      │
│      │
└──────┘
```

You should have been able to watch the computer slowly sketch the square on the screen in a clockwise direction. Remember, you may change the print speed by changing the TO index value for variable J in the time delay loop.

One last problem: how to avoid destroying the square with the READY message.

When the square has been drawn, the cursor is on line 2; when the program ends, the READY message is displayed on the next line, which happens to be within the square. Therefore, before ending the program, you must compensate for this by moving the cursor below the square; the READY message will be written underneath the square and not across it. This is done by printing several CURSOR DOWNs before the END statement.

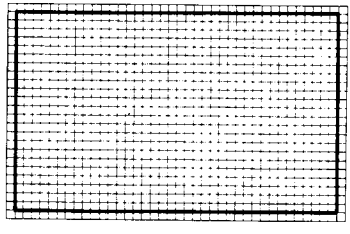
```
50 PRINT"□□□□□□□□":END
```

This will move the cursor down below the square and the square will not be destroyed:

```
┌──────┐
│      │
│      │
│      │
└──────┘
READY.
▒
```

Enlarging the Square

Let's take the small square we just animated and enlarge it so that it forms a boundary one space from the perimeter of a 40-column screen:



If the screen is 40 spaces wide by 25 spaces long, the rectangle's sides should be 38 spaces wide by 23 spaces long:

40−2 (1 space for each side) = 38
25−2 (1 space for each side) = 23

With just a few changes to the animated small square program we can draw a larger rectangle that forms a screen boundary. FOR-NEXT loops were used in the previous animation program to print a string of graphics for each side. To enlarge the square, change the value of the TO index to 36 for the horizontal sides and 21 for the vertical sides, leaving spaces for the corners.

```
15 FOR I=1 TO 36:?"■";
25 FOR I=1 TO 21:?" ■■";
35 FOR I=1 TO 36:?"■";
45 FOR I=1 TO 21:?"■ ■";
```

Horizontal sides

Vertical sides

Make these changes in your program and try a trial RUN.

That was simple. But, because you have created a boundary around the edge of the screen, the last statement of the program (to move the cursor out of the square) is unwanted. Instead, delete line 50 and program the cursor to move inside of the box and print something; you do not want a boundary surrounding an empty screen. Be sure not to program the cursor to go beneath the square, because the screen will scroll up, and you will lose the top of the square. Program something to be printed inside the box, type RUN and watch it go!

THE REAL TIME CLOCK

Another CBM computer feature is the real time clock. The CBM computer clock keeps real time in a 24-hour cycle by hours, minutes, and seconds. The reserved string variable TIME$ or TI$ keeps track of the time.

Setting the Clock

To set the clock, use the following format:

```
TIME$ = "hhmmss"
```

where: hh is the hour between 0 and 23
mm is the minutes between 0 and 59
ss is the seconds between 0 and 59

For hh, enter the hour of the day from 00 (12 AM) to 23 (11 PM). The CBM computer is on a 24-hour cycle so that you can distinguish between AM and PM, unlike 12-hour clocks. The hours from 00 to 11 designate AM, and the hours from 12 to 23 designate PM, returning to 00 at midnight. At midnight, when one 24-hour cycle ends and another begins, hh, mm, and ss are all equal to zero.

When initializing TIME$ to the actual time, type in a time a few seconds in the future. When that actual time is reached, press the RETURN key to set the clock.

```
TIME$="120150"
```

Accessing the Clock

To retrieve the time, type the following in immediate or program mode:

```
?TIME$
```

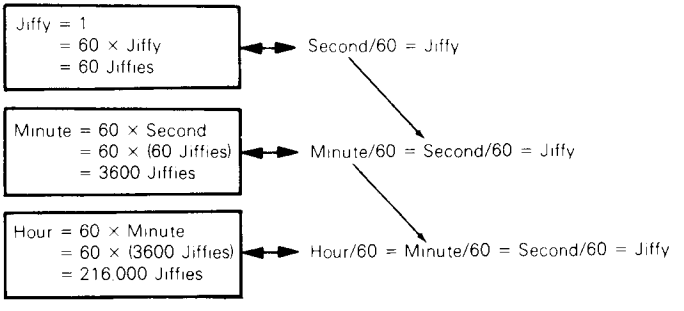
and the computer will display the time in hhmmss:

```
?TIME$
120200
```

The CBM computer clock keeps time until it is turned off. The clock needs to be reset when the computer is powered up again.

Real Time Clock Operation

The CBM computer actually keeps track of time in "jiffies." A "jiffy" is 1/60 of a second. TIME, or TI, is a reserved numeric variable which is automatically incremented every 1/60 of a second. TIME is initialized to zero on start-up, and is reset back to zero after 51,839,999 jiffies. TIME$ is a string variable that is generted from TIME. **When TIME$ is called, the computer displays time in hours, minutes, and seconds (hhmmss), but in fact converts jiffy time to real time.** Notice that TIME$ and TI$ are not the string representations of TIME and TI; they are numbers representing real time, calculated from jiffy time (TIME, TI). The conversion is done as follows. Each second is divided into 60 jiffies. One minute is composed of 60 seconds. One hour is made up of 60 minutes. Therefore one second is 60 jiffies, one minute is 3600 jiffies, and one hour is 216,000 jiffies, as illustrated below:



The following statements convert jiffy time (J) into real time, shown as hours (H), minutes (M), and seconds (S). A complete program follows the statement descriptions.

```
10 J=TI
20 H=INT(J/216000)
```
Calculate hours. Integer function takes only whole number.

```
30 IF H<>0 THEN J=J-H*21600
```
If any hours, subtract number of jiffies in one hour by H to leave remaining jiffies.

```
40 M=INT(J/3600)
```
Calculate minutes. Integer function takes only whole number.

```
50 IF M<>0 THEN J=J-M*3600
```
If any minutes, subtract number of jiffies in minutes by 7 to leave remaining jiffies.

```
60 S=INT(J/60)
```
Calculate seconds. Integer function takes only whole number.

```
 5 PRINT"■REAL TIME":PRINT:PRINT:
10 J=TI
15 T$=TIME$
20 H=INT(J/21600)
30 IF H<>0 THEN J=J-H*21600
```

```
40 M=INT(J/3600)
50 IF M<>0 THEN J=J-M*3600
60 S=INT(J/60)
70 H$=RIGHT$(STR$(H),2)
80 M$=RIGHT$(STR$(M),2)
90 S$=RIGHT$(STR$(S),2)
100 PRINT"H:M:S  ";H$;":";M$;":";S$,"TIME$:  ";T$
110 PRINT"■■■■";:GOTO10
```

In the program above, statements 70 through 90 convert the numeric answers into proper string form for tidy printing. Statement 100 prints both the real time calculated from the program, and TIME$, the real time calculated automatically by the computer. Notice that the result is the same in both cases.

To get an idea of jiffy speed and the conversion from the jiffy to the standard clock, type in the following program; it displays the running time of both TIME$ and TIME (TI):

```
5 REM **RUNNING CLOCKS**
10 PRINT "■REAL TIME:  ":PRINT:PRINT "JIFFY TIME:  "
20 FOR I=1 TO 235959
30 PRINT"■";TAB(13);TIME$
40 FOR J=1 TO 60 STEP 2
50 PRINT"■■■";TAB(12);TI
60 NEXT J
70 NEXT I
```

The FOR-NEXT loop for TIME in line 40 increments by STEP 2 (every two jiffies) for two reasons:

1. Displaying 60 jiffies a second is too fast to read.
2. Displaying each jiffy takes longer than incrementing the jiffy. This delays the loop, so the TIME$ display is slower than it should be. By incrementing and printing every other jiffy we can minimize this delay problem. Run this program and you will see that jiffies increment to 60 within each second. Run this program without STEP 2 in line 40 and see the time delay when printing TIME$.

```
Real time: 006704
Jiffy time: 25500
```

Keeping time in jiffies is useful for timing program speed. This lets you test the efficiency of a program. Consider this short program:

```
10 PRINT"■**KEYBOARD TEST**":PRINT
20 FOR I=32 TO 127
30 PRINT CHR$(I);
40 NEXT I
50 FOR J=161 TO 255
60 PRINT CHR$(J);
70 NEXT J
80 PRINT:PRINT:PRINT"**END TEST**"
```

We can compute execution time for this program as follows:

1. TI (or TIME$) is assigned to a variable constant at the start of the time test.
2. TI (or TIME$) is reassigned to a different variable constant at the end of the time test.
3. Subtract the first TI variable from the second. This will give you the amount of jiffy time it took to process the program that lies in between.

The listing below shows the three added steps:

```
Step 1  10 PRINT"■■KEYBOARD TEST■■":PRINT
        15 A=TI
        20 FOR I=32 TO 127
        30 PRINT CHR$(I);
        40 NEXT I
        50 FOR J=161 TO 255
        60 PRINT CHR$(J);
        70 NEXT J
        75 B=TI
Step 2  80 PRINT:PRINT:PRINT"■■END TEST■■"
Step 3  100 PRINT:PRINT"TI = ";B-A
```

At line 15, variable A is set to the current value of TI.

```
15 A=TI

   A = TI 6001762

   A 6001762
```

Then, as the program is processed, TI increments 60 times every second. At line 75, B is set to the current value of TI.

```
75 B=TI

   B=TI 6001953

   B= 6001953
```

Line 100 subtracts the first value of TI (A) from the second (B).

```
100 PRINT PRINT"TI=";B-A

      B 6001953
    - A 6001762
    ----------
        191
```

The example shows that it took 191 jiffies to print the keyboard characters on the screen. Dividing jiffy time (191 jiffies) by 60 (the number of jiffies in a second):

```
191/60=3.1833
```

shows it took 3.1833 seconds (191 jiffies) to process the program. Below is a sample run of the program.

```
■■KEYBOARD TEST■■

 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFG
HIJKLMNOPQRSTUVWXYZ[\]↑← !"#$%&'()*+,-./
0123456789:;<=>?■■■■■■■■■■■■■■■■■■■■■■■■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

■■END TEST■■

TI = 191
```

Digital Display Clock

The following program is a fun program. It is a variation of the CBM digital clock using enlarged numbers 0 through 9, created with the graphic characters. The program prints out only the hour and minutes due to the size of the screen. The program is long, as you can see, but it is made up almost entirely of PRINT statements to print the numbers. After keying in the program, watch it run.

```
100 PRINT"…";
110 S=INT(TIME/60)
120 M=INT(S/60)
130 H=INT(M/60)
140 M=M-H*60
150 T=H
160 GOSUB500
170 PRINT"…";
180 T=M
190 GOSUB500
200 PRINT"…";
210 GOTO110
500 U=T-10*INT(T/10)
510 T=INT(T/10)
520 D=T+1
530 GOSUB600
540 D=U+1
550 GOSUB600
560 RETURN
600 ON D GOSUB 1000,1100,1200,1300,1400,1500,1600,
    1700,1800,1900
610 RETURN
1000 PRINT"…";
1001 PRINT"…";
1002 PRINT"…";
1003 PRINT"…";
1004 PRINT"…";
1005 PRINT"…";
1006 PRINT"…";
1007 PRINT"…";
1008 PRINT"…";
1009 PRINT"…";
1010 RETURN
1100 PRINT"…";
1101 PRINT"…";
1102 PRINT"…";
1103 PRINT"…";
1104 PRINT"…";
1105 PRINT"…";
1106 PRINT"…";
1107 PRINT"…";
1108 PRINT"…";
1109 PRINT"…";
1110 RETURN
1200 PRINT"…";
1201 PRINT"…";
1202 PRINT"…";
1203 PRINT"…";
1204 PRINT"…";
1205 PRINT"…";
1206 PRINT"…";
1207 PRINT"…";
1208 PRINT"…";
1209 PRINT"…";
1210 RETURN
1300 PRINT"…";
1301 PRINT"…";
1302 PRINT"…";
1303 PRINT"…";
1304 PRINT"…";
1305 PRINT"…";
1306 PRINT"…";
1307 PRINT"…";
```

```
1308 PRINT"◥▚        ▬◤▐■■■■■■■■[▚]";
1309 PRINT" ◥▚     ▬◤    TTTTTTTTT";
1310 RETURN
1400 PRINT"      ▚◤ ▬  ▐■■■■■■■■[▚]";
1401 PRINT"     ▚◤  ▬  ▐■■■■■■■■[▚]";
1402 PRINT"    ▚◤   ▬  ▐■■■■■■■■[▚]";
1403 PRINT"   ▚◤    ▬  ▐■■■■■■■■[▚]";
1404 PRINT"▚◤   ▬◤▚  ▬  ▐■■■■■■■■[▚]";
1405 PRINT"▚  ▬◤ ▚  ▬  ▐■■■■■■■■[▚]";
1406 PRINT"▚          ▬■■■■■■■■■[▚]";
1407 PRINT"▚          ▬■■■■■■■■■[▚]";
1408 PRINT"      ▚ ▬  ▐■■■■■■■■[▚]";
1409 PRINT"      ▚ ▬    TTTTTTTTT";
1410 RETURN
1500 PRINT"▚          ▬■■■■■■■■■[▚]";
1501 PRINT"▚          ▬■■■■■■■■■[▚]";
1502 PRINT"▚  ▬       ▐■■■■■■■■[▚]";
1503 PRINT"▚        ◥▬ ▐■■■■■■■■[▚]";
1504 PRINT"▚         ◥▬■■■■■■■■■[▚]";
1505 PRINT"      ◥▚  ▬■■■■■■■■■[▚]";
1506 PRINT"       ▚  ▬■■■■■■■■■[▚]";
1507 PRINT"▚  ◥▬   ▚◤  ▬■■■■■■■■■[▚]";
1508 PRINT"◥▚        ▬◤▐■■■■■■■■[▚]";
1509 PRINT" ◥▚     ▬◤    TTTTTTTTT";
1510 RETURN
1600 PRINT" ▚◤      ◥▬ ▐■■■■■■■■[▚]";
1601 PRINT"▚◤        ◥▬■■■■■■■■■[▚]";
1602 PRINT"▚  ▬◤  ◥▚  ▬■■■■■■■■■[▚]";
1603 PRINT"▚  ▬       ▐■■■■■■■■[▚]";
1604 PRINT"▚        ◥▬ ▐■■■■■■■■[▚]";
1605 PRINT"▚         ◥▬■■■■■■■■■[▚]";
1606 PRINT"▚  ▬◤  ◥▚  ▬■■■■■■■■■[▚]";
1607 PRINT"▚  ◥▬   ▚◤  ▬■■■■■■■■■[▚]";
1608 PRINT"◥▚        ▬◤▐■■■■■■■■[▚]";
1609 PRINT" ◥▚     ▬◤    TTTTTTTTT";
1610 RETURN
1700 PRINT"▚          ▬■■■■■■■■■[▚]";
1701 PRINT"▚          ▬■■■■■■■■■[▚]";
1702 PRINT"       ▚◤  ▬■■■■■■■■■[▚]";
1703 PRINT"      ▚◤  ▬◤▐■■■■■■■■[▚]";
1704 PRINT"     ▚◤  ▬◤ ▐■■■■■■■■[▚]";
1705 PRINT"    ▚◤  ▬◤  ▐■■■■■■■■[▚]";
1706 PRINT"    ▚  ▬◤   ▐■■■■■■■■[▚]";
1707 PRINT"    ▚  ▬    ▐■■■■■■■■[▚]";
1708 PRINT"    ▚  ▬    ▐■■■■■■■■[▚]";
1709 PRINT"    ▚  ▬       TTTTTTTTT";
1710 RETURN
1800 PRINT" ▚◤      ◥▬ ▐■■■■■■■■[▚]";
1801 PRINT"▚◤        ◥▬■■■■■■■■■[▚]";
1802 PRINT"▚  ▬◤  ◥▚  ▬■■■■■■■■■[▚]";
1803 PRINT"▚  ◥▬   ▚◤  ▬■■■■■■■■■[▚]";
1804 PRINT"◥▚        ▬◤▐■■■■■■■■[▚]";
1805 PRINT"▚◤        ◥▬■■■■■■■■■[▚]";
1806 PRINT"▚  ▬◤  ◥▚  ▬■■■■■■■■■[▚]";
1807 PRINT"▚  ◥▬   ▚◤  ▬■■■■■■■■■[▚]";
1808 PRINT"◥▚        ▬◤▐■■■■■■■■[▚]";
1809 PRINT" ◥▚     ▬◤    TTTTTTTTT";
1810 RETURN
1900 PRINT" ▚◤      ◥▬ ▐■■■■■■■■[▚]";
1901 PRINT"▚◤        ◥▬■■■■■■■■■[▚]";
1902 PRINT"▚  ▬◤  ◥▚  ▬■■■■■■■■■[▚]";
1903 PRINT"▚  ◥▬   ▚◤  ▬■■■■■■■■■[▚]";
1904 PRINT"◥▚          ▬■■■■■■■■■[▚]";
```

```
1905 PRINT" ◥▨          ▄■■■■■■■■■■▮"; 
1906 PRINT"          ▨  ▄■■■■■■■■■■▮";
1907 PRINT"▨   ▀   ◢  ▄■■■■■■■■■■▮";
1908 PRINT"◥▨         ◢■■■■■■■■■■■▮";
1909 PRINT" ◥▨        ◢    TTTTTTTTTT";
1910 RETURN
```

RANDOM NUMBERS

Random numbers are generated by the CBM computer using an algorithm that depends on a starting number, or seed. The same seed always generates the same sequence of random numbers.

RANDOM NUMBER SEED

Every CBM computer has a constant initial seed number which it generates when power is first turned on. This initial seed number will probably differ from one CBM computer to the next, but for any single CBM computer the same seed is generated whenever power is turned on. Therefore, for any single CBM computer, the same initial sequence of random numbers will be generated each time the computer is powered up. The display below shows the first five numbers of a typical sequence, as it might appear when the RANDOM function RND(arg) is executed after power-up:

```
### COMMODORE BASIC ###

 7167 BYTES FREE

READY.

FOR I=1 TO 5:? RND(1):NEXT
RUN
 .880969862
 .355265655
 .659512252
 .803285178
 .546991144
READY.
```

CBM computer random function logic is best visualized as accessing a large number of fixed random number sequences. These random number sequences will vary from one CBM computer to the next, but they will always be the same for any given CBM computer.

Each random number sequence is identified by a seed, which is a negative number. This may be illustrated as follows:

Four random number seeds

-1	-2	-3	-4
.592222864	.529448248	.619245849	.618803331
.217940608	.217131897	.217986376	.215972203
.0371848992	.125471999	.12554828	.126128101
.867675019	.869597673	.867369876	.869521353
.805311997	.805998629	.808165423	.807814458

First five random numbers in each list

Every negative number seeds a different random number list. There are innumerable negative numbers, therefore there are innumerable lists of random numbers which can be accessed by any CBM computer.

You select any random number sequence by executing any BASIC statement that includes the RND function with a negative argument. You can use a simple assignment statement such as:

```
20 X=RND(-2)
```

Executing a BASIC statement that includes an RND function with a negative argument has the effect of resetting a pointer to the first random number in that negative argument's random number list. For example, on one particular CBM computer, executing the assignment statement on line 20 above will reset the random function pointer to the number .271819872, the first number in the list seeded by −2. This reset will occur every time the RND(−2) function is encountered. The CBM computer that was used to generate this particular example selects the number .271819872, but another CBM computer will have a totally different fixed random number sequence initialized by the (−2) seed. If you have three CBM computers, each will have a different random number sequence initialized by the (−2) seed; however, each CBM computer will initialize to the same random number sequence on encountering an RND(−2) function.

Random Number Sequences

Having initialized the random number generator to the first number in a particular list, you access sequential random numbers in the list by executing any BASIC statement that includes an RND function with a positive argument. Here, for example, is a program that will display the first six numbers of five random number sequences, seeded by the negative functions −1 through −5:

```
30 FOR I=-1 TO -5 STEP -1
35 X=RND(I):PRINTI
40 FOR J=1 TO 5
50 PRINT RND(1)
60 NEXT J
70 NEXT I
100 STOP
```

The random function on line 35 occurs in the outer FOR-NEXT loop; it resets the random function generator pointer to the first element of five different sequences for the five negative values of I: −1, −2, −3, −4, and −5. The inner FOR-NEXT loop (lines 40, 50, and 60) displays the first six elements in each of these five random number sequences.

```
-1
 .592222864
 .217940608
 .0371848992
 .867675019
 .805311997
-2
 .529448248
 .217131897
 .125471999
 .869597673
 .805998629
```

```
-3
 .619245849
 .217986376
 .12554828
 .867369876
 .808165423
-4
 .618803331
 .215972203
 .126128101
 .869521353
 .807814458
-5
 .529738186
 .216918235
 .128416908
 .868422708
 .717787951
```

To demonstrate the existence of a fixed number sequence in each random list, stop the generation of numbers for a list and then restart it. Look at the following modification of the random number generator program:

```
30 FOR I=-1 TO -5 STEP -1
35 X=RND(I):PRINTI
40 FOR J=1 TO 3
50 PRINT RND(1)
60 NEXT J
70 FOR K=10 TO 11
75 PRINT RND(1)
80 NEXT K
90 NEXT I
100 STOP
```

This program again references the first six elements in five random number lists, but it does so in two separate FOR-NEXT loops. Nevertheless, when you execute this program you will get exactly the same display as the earlier program. In other words, it does not matter where or how a random function with a positive argument is executed, it will always access the next element of the fixed random number list identified by the most recent negative seed. Moreover, any time this negative seed is encountered in a subsequent random function, the list pointer immediately returns to the first element of the sequence. For example, add this statement to the program shown above:

```
65 X=RND(I)
```

Now when you execute the program, you will access the first three numbers in each list, then the first two numbers in each list will be re-accessed.

Now experiment by keying in the programs illustrated above. Vary both the negative seed numbers specified by the I index in the outer FOR-NEXT loop, and the number of elements selected by J and K in the inner FOR-NEXT loops. Experiment in this fashion until you are completely satisfied that you understand the manner in which random numbers are generated.

Printing Random Numbers

In order to better compare random numbers, you should print results rather than displaying them (assuming you have a printer). Although printer programming is de-

scribed in Chapter 6, necessary additional statements are shown below in order to select a printer.

```
10 OPEN 4,4
20 CMD 4
30 FOR I=-1 TO -5 STEP -1
35 X=RND(I):PRINTI
40 FOR J=1 TO 5
50 PRINT RND(1)
60 NEXT J
70 NEXT I
80 PRINT#4
90 CLOSE 4
100 STOP
```

Statements on lines 10 and 20 select the printer; statements on lines 80 and 90 deselect it. Make sure the printer power is turned on, and that it is connected correctly to your CBM computer; then use variations of the program illustrated above to experiment with random numbers. The hard copy printed with each experiment will make it easier for you to compare the numbers generated by different variations of your program.

If you execute a statement that contains an RND function with a 0 argument, then the random number generated depends on the system clock. But there are similarities in random number patterns generated by sequences of RND functions with a 0 argument. To prove this to yourself, change the argument of the RND function on line 50 from 1 to 0 and reexecute the random number generator program a few times. You will see that no two sequences of numbers are identical, but they certainly are quite close.

Random Seeds

To generate a totally different random number you need to have some way of generating a totally random seed. This can be done using the current jiffy count, TI:

```
10 X=RND(-TI): REM START SEED
```

Now you will get a different random sequence started each time statement 10 is executed.

A more nearly pure random seed can be obtained by using RND (−RND(0)), but only if your CBM computer has the new BASIC ROMs. For example:

```
10 X=RND(-RND(0))
```

Here again you will get a different random sequence started each time statement 10 is executed.

In the programs that follow, −TI is used, as it is compatible with both the old and new ROMs. If you have the new ROMs, you can use −RND(0) in place of −TI.

Generating Random Dice Throws

Random numbers are initially generated in the range 0 through 1. You will have to convert the random number to whatever range you require. Suppose numbers must range from 1 to 6 (as in one die number of a dice game). You will need to multiply the random number by 6:

```
6*RND(1)
```

This gives a floating point number in a range just greater than 0 but just less than 6 ($0<n<6$). Add 1 to get a number in the range $1<n<7$:

```
6*RND(1)+1
```

Then convert the number to an integer, which discards any fractional part of a number, returning the number in the range 1 to 6 but in integer form:

```
INT(6*RND(1)+1)
```

or:

```
A%=6*RND(1)+1
```

The general cases for converting the RND fraction to whole number ranges are shown below. Note that the INT function will only handle numbers in the integer range ±32767.

```
INT((n+1)*RND(1))          Range 0 to n
INT(n*RND(1)+1)            Range 1 to n
INT((n-m+1)*RND(1)+m       Range m to n
```

Now experiment with a variety of different random number ranges by modifying the statement(s) illustrated above.

The program below shows −TI being used to generate a random seed. This program calculates numbers in the range m to n; in this program, the values of m and n are set in line 10 for a given program run. Note that these values can be negative. In the following example, the display is an unending sequence of random numbers between −50 and +50. (Press the STOP key to end the program.) A different sequence of numbers will be printed each time the program runs, since −TI provides a random seed. Note that the X value returned from RND(−TI) is displayed instead of the TI value.

```
10 M=-50:N=50
20 X=RND(-TI):PRINT X
30 FOR I=1 TO 8
40 C%=(N-M+1)*RND(1)+M
50 PRINT C%;:NEXT I
60 PRINT:GOTO 30
RUN
 8.27633085E-06
-14  9 -34 -35 -47 -44  28  31
 29 -8 -36 -28 -42 -28  15  14
 7 -13  3 -8  8  41  19 -43
 35  12  24 -7 -7 -21 -47  1
-32 -49  7 -49  28 -22 -17 -24
-12  7  27  1  11  9 -18  35
 48  49  1  34 -46 -29 -43  29
-18  5 -30  2  8 -28 -13 -23
 48 -15 -12 -45  26  44 -25  2
-9  4  27  50  33 -16 -43 -15
 20  20  17  43 -18 -48 -38  24
-16  43 -50  36 -38  5  11  25
-30  6 -25 -47  32  10  42 -21
-47 -38 -28 -8  16 -20  42 -4
-34  36 -17  27 -8 -49 -6 -35
-19  19 -35  48 -42  36 -25  2
-49  37  47  38 -20 -25  32 -50
-5 -35 -35  17 -41  36 -19  4
 33 -20  45 -7  48 -4 -33 -10
 1  27 -39 -14 -38 -6  4  10
-5  17  2  49  0 -40 -5  32
-50  32 -24 -37 -38  22 -13 -27
-24 -30  35  10  6  16 -50  49
-49  50  43  38 -21  47 -43  28
 32 -35 -18 -5  27 -46 -14  23
-49 -45  27  7 -35  1  46 -25
-8  20 -8 -12 -46 -31 -17 -18
-47  47 -49  18  47  17  40 -13
-40  48 -41 -33  5 -14 -46  45
-29 -37  22  17  42  33 -31  49
 8 -4  36  37  11  18  29  25
 0 -1  2 -16  32 -29 -31  33
-9 -41 -4  47  12 -22  9 -48
-40  32  15  32 -50  3 -9  19
```

To illustrate different number ranges, change the values of M and N in line 10 of the above program. For example, make M=1 and N=6; this will generate and unending sequence of random numbers between 1 and 6.

Random Selection of Playing Cards

A quick scan of the display above shows that numbers repeat within the first 100 generated. That is, every 101 numbers will not pick a number in the range −50 to +50 with every number present and no duplications. This is fine in, say, a dice game where you take the rolls as they come. For other random number uses, however, **you may need to develop random numbers in a certain range where every number is accounted for, and there are no duplications. An example is dealing from a deck of cards.** You need to pick a card, and when that card has been picked it cannot be picked again during the same deal.

The program below shows one way to program shuffling a deck of cards on the CBM computer. This program fills a 52-element table D% with the numbers 1 through 52 in a random sequence. (Element D%(0) is not used.) The cards can be pegged to the random numbers in any way, such as:

A=1, 2=2, 3=3, ..., Q=12, K=13
Spades=0, Hearts=13, Diamonds=26, Clubs=42

With this scheme the Ace of Spades=1+0=1, the Queen of Spades=12+0=12, the Three of Hearts=3+13=16, etc.

In the shuffle program, a 52-element flag table FL keeps track of whether a card has been picked or not. PRINT statements are inserted to display the seed value, followed by the numbers, in a continuous-line format. Note that exactly 52 numbers are displayed and that no number is repeated. Each program run will produce a new random sequence.

```
10 DIM FL(52),D%(52)
20 X=RND(-TI):PRINT X
30 FOR I=1 TO 52
40 C%=52*RND(1)+1
50 IF FL(C%)<>0 GOTO 40
60 D%(I)=C%:FL(C%)=1
70 PRINT C%;
80 NEXT I
RUN
 1.18586613E-05
 48  40  13  37  50  43  46  31  49  44
 23  38  25  11  9  35  32  30  24  41
26  5  6  1  45  10  21  14  42  20  15
 34  18  52  47  7  16  8  19  33  36  4
  17  3  22  27  29  28  39  2  51  12
RUN
 1.01154728E-06
 14  35  52  50  26  48  27  36  34  25
 18  20  41  33  39  7  46  24  23  28  1
9  3  12  43  2  31  44  4  1  32  37  3
0  40  22  45  48  42  49  16  11  6  10
  29  9  51  17  8  15  38  5  21  13
```

But this program runs more slowly as it nears the 52nd number. It is especially slow on the last card. This is because the program has to fetch more and more random numbers to find one that has not already been picked. A simple routine such as this has much room for improvement, of course. It can be speeded up just by findng the last number in the program from the table rather than waiting until it is selected randomly.

RANDOM POKE TO THE SCREEN

The following program is a modification of program BLANKET. Instead of displaying a character in continuous-line format, this program fills the screen by randomly POKEing the character into the 1000 positions of the screen.

Here is the first version.

```
10 REM *******  B L A N K E T  *******
20 REM RANDOM DISPLAY OF ONE
30 REM   CHARACTER ENTERED FROM THE
40 REM   KEYBOARD
50 REM ********************************
90 PRINT"HIT A KEY OR <R> TO END";
100 GET C$:IF C$="" GOTO 100
105 IF C$=CHR$(13) GOTO 170
110 PRINT"□";          :REM CLEAR SCREEN
120 X=RND(-TI)         :REM START NEW SEED
125 C=(ASC(C$)AND128)/2 OR (ASC(C$)AND63)
127 A=1000*RND(1)+32768
130 POKE A,C           :REM DISPLAY CHAR
140 GET D$:IF D$="" GOTO 127
150 C$=D$
160 GOTO 105
170 END
```

The program is the standard BLANKET program through line 110, where a new character is input and the screen is cleared. The statement on line 120 stores a new seed in preparation for a random display sequence on the screen. The statement on line 125 converts C$ to its equivalent POKE number. The statement on line 127 calculates a random screen address in the range 32768 to 33767; using the RND range formula with m = 32768 and n = 33767 as follows:

```
(n-m+1)*RND(1)+m                    Range formula
(33767-32768+1)*RND(1)+32768        as used in line 127
=1000*RND(1)+32768
```

Neither the INT function nor an integer variable (which would have been A%) can be used, because the screen addresses begin just beyond the maximum integer value of 32767. Fortunately the POKE function, which is where the screen addresses will be used, simply discards any fractional portion of a real number address presented to it. (For other applications when you are dealing with random numbers outside the integer range, you will have to check that the floating point equivalent provides the intended range.)

The first version of the program above randomly fills the screen with the keyed-in character. It does this by simply POKEing to random screen locations. It may POKE many times to the same location when other locations are not yet filled, and it continues to POKE, even after the screen is filled, until a new character is keyed in.

When the program is run, about half the screen positions quickly fill with the character. Then character placement slows down more and more until at the end, when the screen is almost filled, and remaining positions are filled very slowly. It takes about three minutes to completely fill the screen with this version of the program.

The program is operating at the same speed throughout, but it does not get much work done towards the end, because many of the positions that it POKEs to are already filled. The program appears to slow down because displaying a character over the same character has no visible effect.

The program can be speeded up a good deal by eliminating the superfluous POKEs to screen positions that are already filled. A new version of program BLANKET does this.

Rather than calculating a number in the same range all the time and discarding, or in this case re-POKEing, the duplicate numbers, the new program decreases the range of numbers generated to correspond with the number of items left to operate on. It does this by keeping track in a table of the screen positions remaining to be filled, and generating a random number within the range of table indexes yet to be POKEd to. The POKE address itself is retrieved from the contents at the table index.

```
5 REM RANDOM VERSION 2
10 REM *******  B L A N K E T  *******
20 REM RANDOM DISPLAY OF ONE
30 REM   CHARACTER ENTERED FROM THE
40 REM   KEYBOARD
50 REM ********************************
70 DIM T(999)
80 GOSUB 200     :REM INITIALIZE TABLE
90 PRINT"HIT A KEY OR <R> TO END";
100 GET C$:IF C$="" GOTO 100
105 IF C$=CHR$(13) GOTO 170
110 PRINT"■";      :REM CLEAR SCREEN
120 X=RND(-TI)     :REM START NEW SEED
125 C=(ASC(C$)AND128)/2 OR (ASC(C$)AND63)
126 FOR N=999 TO 0 STEP -1
127 A%=(N+1)*RND(1) :REM PICK AN ELEM
128 A=T(A%)+32768  :REM FORM POKE ADDR
129 TP=T(A%):T(A%)=T(N):T(N)=TP :REM SWAP ELEMENTS
130 POKE A,C       :REM DISPLAY CHAR
140 NEXT N
160 GOTO 100
170 END
199 REM **SUBR TO INITIALIZE TABLE**
200 FOR I=0 TO 999:T(I)=I:NEXT
210 RETURN
```

In this program, the table holds the 1000 screen position indicators; it is dimensioned on line 70.

At line 80 an initialization subroutine is called that places the numbers 0 through 999 into corresponding elements of Table T. T(0) will contain 0, T(1) will contain 1,. . .T(999) will contain 999. The elements do not have to be filled with consecutive numbers since they are to be picked randomly, but this is the easiest way to program the fill loop. In fact, the table will be in order only the first time the program is run after loading. Lines 90 through 125 hold exactly the same program statements as in the earlier version.

Lines 126 through 140 hold a FOR-NEXT loop that fills the 1000 screen locations with the keyed-in character. The statement on line 127 picks a random table index A% from the remaining unfilled range of 0 to N. The expression (N+1)*RND(1) performs this task. The statement on line 128 forms the POKE address A as the sum of the T table element whose index was picked on line 127, plus the beginning screen memory address of 32768. The statement on line 129 exchanges the chosen table element T(A%) with the highest active table element T(N) via a temporary location TP. The statement on line 130 displays the character at the random screen location. The NEXT N at line 140 decrements the pointer N so that the used screen address just swapped into T(N) is not picked again during the current program run.