Chapter 5

Making the Most of CBM Features

This chapter describes CBM computer hardware characteristics and programming techniques.

HARDWARE FEATURES

KEYBOARD ROLLOVER

If you press two or more keys simultaneously, or if you press a second key before the first character is displayed, a keystroke will be ignored — unless your keyboard has "rollover." Rollover "remembers" a keystroke until it is displayed. Fortunately, CBM computer keyboards have rollover.

Rollover remembers incoming keystrokes while a preceding keystroke is being processed. The "remembered" keystrokes are stored in a buffer until they are processed. Without this buffer, rapidly incoming keystrokes would be lost. For example, if keystroke #2 occurs before keystroke #1 has been processed, the CBM computer stores keystroke #2 in the buffer until keystroke #1 has been processed. Then keystroke #2 is taken from the buffer and processed in turn.

Rollover is a very useful feature of the CBM computer keyboard; it allows you to type in data very fast without the loss of occasional keystrokes.

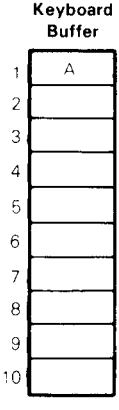
KEYBOARD BUFFER

All CBM computers have a 10-character buffer that holds characters when keys are pressed at the keyboard.

To illustrate, load and run the final version of the BLANKET program, listed in Figure 5-1. Press a key. While the first display is generated, press up to ten more keys, then sit back and relax. Each of the ten keyed-in characters will be fetched from the buffer in turn and displayed by the BLANKET program.

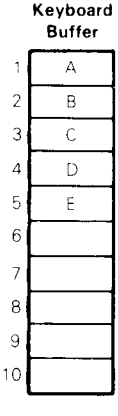
Let us look at this process in more detail.

Whenever you press a key, it goes into the first storage location in the 10-character keyboard buffer. If you press the A key, this is what happens:



The CBM computer keeps track of the number of characters in the buffer and the location of the next character to be displayed. Each time the GET statement fetches another character, a buffer pointer is incremented to select the next buffer location.

If you press additional keys while the A is being displayed, the additional characters are stored in the keyboard buffer beginning at the next available location. Suppose you type in A, and while A is being displayed you type in B, C, D, and E. These characters are all stored in the keyboard buffer:

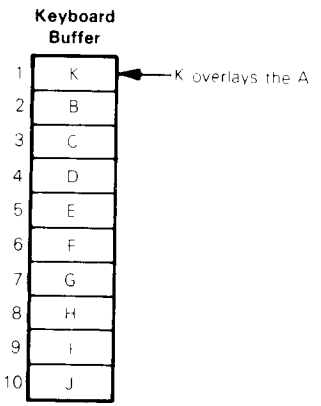


```
10 REM *******  B L A N K E T  *******
20 REM CONTINUOUS-LINE DISPLAY OF ONE
30 REM   CHARACTER ENTERED FROM THE
40 REM   KEYBOARD
50 REM ********************************
90 PRINT"HIT A KEY OR <R> TO END";
100 GET C$:IF C$="" GOTO 100
105 IF C$=CHR$(13) GOTO 170
110 PRINT"□";          REM CLEAR SCREEN
120 FOR I=1 TO 920     REM 920/40=23 LINES
130 PRINT C$;
140 NEXT
150 PRINT"PHEW!"
160 GOTO 90
170 END
```

Figure 5-1. Program BLANKET

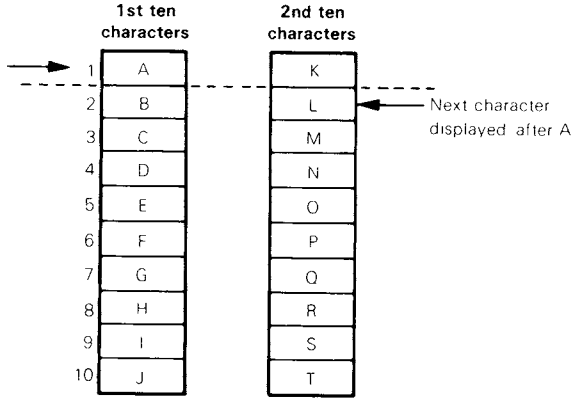
If you let the BLANKET program continue to run, it will successively display all the letters stored in the keyboard buffer. After A is finished, the program fetches B and displays it across 20 lines, then it fetches C and displays it, etc.

If you type in more than ten characters, then for any model with the exception of the 8000 series, the buffer pointer wraps around, returning to buffer position 1. For example, if you type in the first 11 letters of the alphabet (A-K), the first ten letters are stored in the ten buffer locations, then the letter K is stored in the first buffer location, overlaying the A:



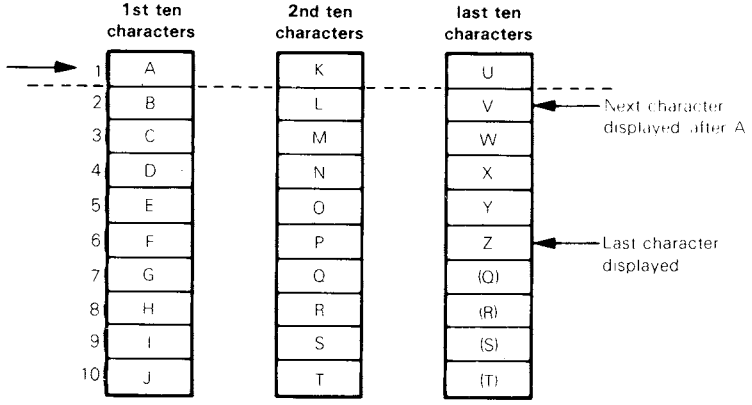
When the program finishes displaying the A, it returns to fetch another character. But the CBM computer has already fetched the character in location 1, so it considers the buffer empty. Keying in exactly eleven characters, or multiples of eleven characters, produces no additional automatic displays in program BLANKET.

Typing in 12 to 20 characters displays the first character, and then a string of characters beginning with character 12. For example, type in A. While A is being displayed type in B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, and T.



The order of display is: A, L, M, N, O, P, Q, R, S, T.

This logic holds true for additional multiple characters. Type in A, and while A is being displayed type in the rest of the alphabet. (You will have to be quick to do this.)



A negating effect occurs every 11 characters. For instance, type in A and B, and let A display completely. Then, while B is displaying, type in C, D, E, F, G, H, I, J, K, and L. The additional ten characters are cancelled out, just as the additional ten characters B through K were when entered while A was being displayed.

The CBM 8000 discards input characters which other models wrap around within the input buffer.

Emptying the Buffer Before a GET

The keyboard buffer is a mild surprise, usually a pleasant one. For program BLANKET you can save up the characters you want displayed rather than keying them in one at a time in response to the HIT A KEY message. But the keyboard buffer can also come as a rude shock. **Accidentally pressing a key may cause a program to fetch an unwanted character from the keyboard buffer. To avoid this, you can program a loop to empty the keyboard buffer** before fetching an intended response character as follows:

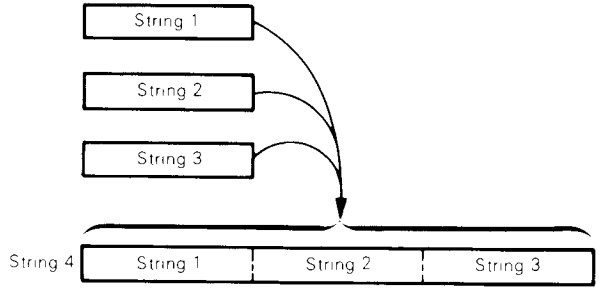
```
95 FOR I=1TO10 GET C$:NEXTI  REM EMPTY KYBD BFR
100 GET C$ IF C$="" GOTO 100
```

The statements on line 95 empty the keyboard buffer by getting all ten possible buffer characters.

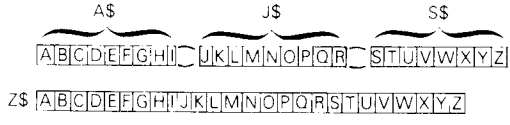
Edit program BLANKET by adding line 95 as shown above. Now press any combination of keys while a character is being displayed. Any stored characters are fetched and discarded by the GET loop, so you will not have any automatic continuous display.

STRING CONCATENATION

Within strings the CBM computer will accept alphabetic, graphic, and numeric characters, or combinations of these. While handling strings, it may be useful to create a single string by linking shorter strings end to end in a chain-like fashion:



Suppose, for example, we want to create one large string, Z$, containing the alphabet A through Z. To do this we can link together the last character of A$, shown below, to the first character of J$, and the last character of J$ to the first character of S$, as follows:



The arithmetic operator "+" adds the contents of numeric variables, but when used with strings the "+" concatenates the strings. Table 5-1 summarizes the effect of the "+" operator on strings and numbers.

Table 5-1. Addition (+) Operations

Sign	Type	Example Statement.	Operation	Result
+	numbers	P = 2 + 3	2 + 3	P = 5
+	numeric variables	Q = T + S T = 1 2 3 4 5 S = 1 1 1 1 1	12345 +11111 23456	Q = 23456
+	alphabetic strings	R$ = A$ + F$ A$ = A B C D E F$ = F G H I J	A B C D E ⁀ F G H I J	R$ = A B C D E F G H I J
+	numeric strings	Q$ = T$ + S$ T$ = 1 2 3 4 5 S$ = 1 1 1 1 1	1 2 3 4 5 ⁀ 1 1 1 1 1	Q$ = 1 2 3 4 5 1 1 1 1 1

A word of caution: strings cannot be separated or broken apart in the same fashion as they are concatenated; they cannot be "subtracted" the way they are "added." For instance, to create string X$ containing the contents of J$ and S$ from our original strings A$, J$, S$, and Z$, it would be incorrect to type:

```
X$=Z$-A$ ←——— Incorrect
```

Try it yourself. Enter the values of A$, J$, S$, and X$=Z$−A$ into the CBM computer as shown below. The computer will respond with a ?TYPE MISMATCH ERROR IN LINE 50.

```
10 A$="ABCDEFGHI"
20 J$="JKLMNOPQR"
30 S$="STUVWXYZ"
40 Z$=A$+J$+S$
50 X$=Z$-A$ ←——— Incorrect attempt to get J through Z string
60 PRINT X$

RUN

?TYPE MISMATCH ERROR IN LINE 50
```

The only valid arithmetic operator for strings is the addition sign (+). The other arithmetic operators (−,*,/) will not work, although the Boolean operators (<, >, =) may be used for string comparison.

The correct method of extracting part of a larger string is to use string functions. With the LEFT$, MID$, and RIGHT$ functions it is possible to extract any desired portion of a string. In our example, the letters J through Z can be extracted as follows:

```
50 X$=RIGHT$(Z$,17)
   X$ = RIGHT$(A B C D E F G H I J K L M N O P Q R S T U V W X Y Z ,17)
   X$ = J K L M N O P Q R S T U V W X Y Z
```

or the string may be built by concatenating J$ and S$:

```
50 X$=J$+S$
   X$ = J K L M N O P Q R + S T U V W X Y Z
   X$ = J K L M N O P Q R S T U V W X Y Z
```

Printer/Screen Concatenation

If you want to concatenate strings for screen or printer output only, **use the PRINT statement with semicolon separators (;)** between the strings:

```
PRINT A$;J$;S$

ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

The screen result (A through Z) is not retained anywhere in CBM computer memory.

GRAPHIC STRINGS

Graphic strings are concatenated in the same way as alphabetic strings. This is a useful way of creating pictures and diagrams.

NUMERIC STRINGS

A numeric string is a string whose contents can be evaluated as a number. Numeric strings may be created in two different ways, each yielding slightly different results.

When numeric variables are assigned to numeric strings using the STR$ function, the sign value preceding the number (blank if positive, "−" if negative) is transferred along with the number. This is shown in the short program below:

```
10 AB=12345
20 T$=STR$(AB)
30 PRINT"AB=";AB
40 PRINT"T$=";T$

RUN

AB= 12345
T$= 12345
```

However, **if a number is entered enclosed within quotation marks, or if the number is entered as a string with an INPUT, GET or READ statement,** then the numeric string is treated like any other alphabetic or graphic string. **No blank for a positive sign value is inserted before the number.** This is demonstrated in the following program:

```
10 AB=12345
20 T$="12345"
30 PRINT"AB=";AB
40 PRINT"T$=";T$

RUN

AB= 12345 <---- Space inserted
T$=12345 <---- No space inserted
```

Let us now concatenate two numeric strings, T$ and Q$, to make a new numeric string W$. W$ is to contain the ten digits 1, 2, 3, 4, 5, 6, 7, 8, 9, 0. Here is one possibility:

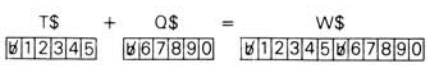
```
10 T=12345
20 Q=67890
30 T$=STR$(T)
40 Q$=STR$(Q)
50 W$=T$+Q$ <------ Create new string W$
60 PRINT"W$=";W$

RUN

W$= 12345 67890
```

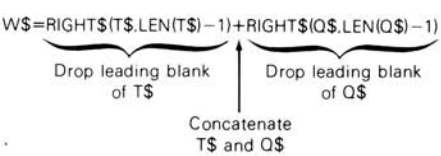
Why the blanks before the 1 and 6? T$ and Q$ were originally positive numeric variables T and Q; when T and Q were converted from numbers into strings, the blank sign position was transferred along with the number.

Therefore, when T$ and Q$ are concatenated, the new string W$ contains a first-digit blank, and an embedded blank before the first digit of Q$.



To get rid of the embedded blanks go back to the separate strings T$ and Q$. Look again at the contents of T$ and Q$ above. The only values we want in W$ are the numbers to the right of the sign value in both T$ and Q$. With the LEFT$, MID$, and RIGHT$ commands you can select any character or group of characters from within a given string. We want all the characters to the right of the first character, the first character being the sign value (either blank or "−"). T$=RIGHT(T$,LEN(T$)−1) does the trick:

Since the first digit needed is in the second position of the string, we tell the CBM computer to use only the values starting in position #2. We can concatenate T$ and Q$ and drop the leading blanks all in one statement:



Our example program, amended to eliminate the sign digits, appears as follows:

```
10 T=12345
   T = 12345
20 Q=67890
   Q = 67890
30 T$=STR$(T)
   T$ = 12345
40 Q$=STR$(Q)
   Q$ = 67890
50 W$=RIGHT$(T$,LEN(T$)-1)+RIGHT$(LEN(Q$)-1)
   W$ = RIGHT$(T$,6-1)          +RIGHT$(Q$,6-1)
   W$ = RIGHT$(T$,5)            +RIGHT$(Q$,5)
   W$ = T$ 12345                +Q$ 67890
   W$ = 1234567890
60 PRINT "W$=";W$

RUN

W$=1234567890
```

In the example above, note that line 50 does not check for negative numbers. If both numbers are negative, then the leading character of T$ should not be dropped; this allows the negative sign to appear in front of the entire number W$. If the two strings have different signs, they should not be concatenated.

INPUT AND OUTPUT PROGRAMMING

The beginning programmer quickly discovers that the input and output sections of a program are its trickiest parts.

Nearly every program uses data which must be entered at the keyboard. Will a few INPUT statements suffice? In most cases the answer is no. What if the operator accidentally presses the wrong key? Or worse, what if the operator discovers that he or she input the wrong data — after entering two or three additional data items? **A usable program must assume that the operator is human, and will likely make every conceivable human error.**

Results, likewise, cannot simply be displayed, or printed, by executing a bunch of PRINT statements. A human being will have to read this output. Unless the output is carefully designed, it will be very difficult to read; as a consequence information could be misread, or entirely overlooked.

Fortunately CBM BASIC has many capabilities that make it easy to program input and output correctly. We will describe some of these capabilities before looking specifically at good input and output programming practices.

PRINT STATEMENT

Semicolon Punctuation

Normally a PRINT statement ends its display with a RETURN. This causes the next PRINT statement to begin displaying in the first character position of the next line. Thus the following immediate mode program displays a column of 20 characters in the first character position of 20 rows:

```
C$="W":FOR I=1 TO 20:? C$:NEXT:?"PHEW!"
W
W
W
W
W
W
W
W
W
W
W
W
W
W
W
W
W
W
W
W
PHEW!

READY.
▒
```

A semicolon (;) appearing after any variable in the PRINT statement causes the next display to begin immediately at the next available character position. A semicolon following the last (or only) variable in the PRINT statement parameter list suppresses the RETURN. Therefore the following program will display 800 characters across 20 rows of a 40-column display.

```
C$="W":FOR I=1 TO 800:? C$;:NEXT:?"PHEW!"
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
PHEW!

READY.
```

The FOR-NEXT loop index I is used as a counter to indicate the number of W's to be displayed, in this case 800. On the first PRINT, a new line is begun and the character W is displayed. The semicolon prevents a RETURN to the next line, so the cursor remains at the character position following the first W. The second W is then displayed and the cursor is left in the next character position. This sequence continues up to the end of the first line, then the cursor moves to the beginning of the next line. This sequence continues for 20 lines (of a 40-column display).

Why does PHEW! print on a new line? It doesn't really; it appears to start a new line because the last character is displayed in the last position of the previous line. Change 800 to 780 and PHEW! is displayed at the end of the line of characters. This may be illustrated as follows:

```
C$="-":FOR I=1 TO 780:?C$;:NEXT:?"PHEW!"
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
--------------------PHEW!

READY.
```

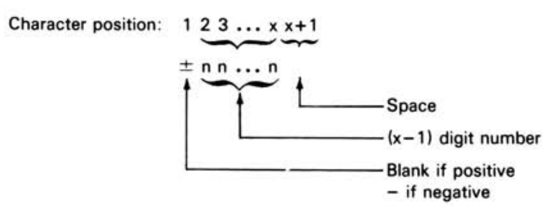
The semicolon concatenates string data, displaying items right next to each other, with no spaces in between. **Numeric data is also displayed in a continuous line format, but with a single space between negative numbers and two spaces between positive numbers** (since the + sign is not displayed).

To illustrate this, change the string variable to a single-digit numeric variable. Three character positions are needed to display each number, so change the ending index to 800/3=267. The number 5 is displayed as follows:

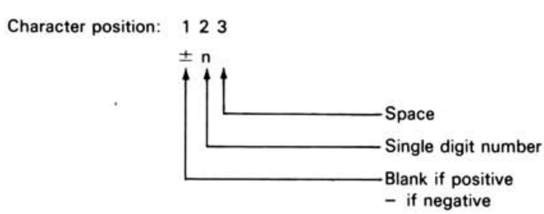
```
C=+5:FOR I=1 TO 267:?C;:NEXT:?"PHEW!"
 5  5  5  5  5  5  5  5  5  5  5  5  5
5  5  5  5  5  5  5  5  5  5  5  5  5  5
  5  5  5  5  5  5  5  5  5  5  5  5  5
 5  5  5  5  5  5  5  5  5  5  5  5  5
5  5  5  5  5  5  5  5  5  5  5  5  5  5
  5  5  5  5  5  5  5  5  5  5  5  5  5
 5  5  5  5  5  5  5  5  5  5  5  5  5
5  5  5  5  5  5  5  5  5  5  5  5  5  5
  5  5  5  5  5  5  5  5  5  5  5  5  5
 5  5  5  5  5  5  5  5  5  5  5  5  5
5  5  5  5  5  5  5  5  5  5  5  5  5  5
  5  5  5  5  5  5  5  5  5  5  5  5  5
 5  5  5  5  5  5  5  5  5  5  5  5  5
5  5  5  5  5  5  5  5  5  5  5  5  5  5
  5  5  5  5  5  5  5  5  5  5  5  5  5
 5  5  5  5  5  5  5  5  5  5  5  5  5
5  5  5  5  5  5  5  5  5  5  5  5  5  5
  5  5  5  5  5  5  5  5  5  5  5  5  5
 5  5  5  5  5  5  5  5  5  5  5  5  5
5  5  5  5  5  5  5  5  5  5  5  5  5  5
 PHEW!

READY.
```

Note the single space between the last number displayed and the word PHEW! This is because numbers are displayed using the following format:



which for a single digit becomes:



Multiple-digit numbers will scroll the display off the screen unless the TO index is adjusted. If C is changed to 2001, a 6-digit display field is needed; you should adjust the TO index from 267 to 800/6=134:

```
C=2001:FOR I=1 TO 134:?C;:NEXT:?"PHEW!"
 2001  2001  2001  2001  2001  2001  200
1  2001  2001  2001  2001  2001  2001  2
001  2001  2001  2001  2001  2001  2001
 2001  2001  2001  2001  2001  2001  200
1  2001  2001  2001  2001  2001  2001  2
001  2001  2001  2001  2001  2001  2001
 2001  2001  2001  2001  2001  2001  200
1  2001  2001  2001  2001  2001  2001  2
001  2001  2001  2001  2001  2001  2001
 2001  2001  2001  2001  2001  2001  200
1  2001  2001  2001  2001  2001  2001  2
001  2001  2001  2001  2001  2001  2001
 2001  2001  2001  2001  2001  2001  200
1  2001  2001  2001  2001  2001  2001  2
001  2001  2001  2001  2001  2001  2001
 2001  2001  2001  2001  2001  2001  200
1  2001  2001  2001  2001  2001  2001  2
001  2001  2001  2001  2001  2001  2001
 2001  2001  2001  2001  2001  2001  200
1  2001  2001  2001  2001  2001  2001  2
001 PHEW!

READY.
```

Numbers are broken across the end of lines. This is because the semicolon (;) generates a continuous display and nothing but an end of line can cause a return.

Comma Punctuation

Commas appearing after a variable, or at the end of a PRINT statement, treat the display as though it were tabbed at ten-character intervals. For a 40-column display this may be illustrated as follows:

```
1         11        21        31
↑
└──────── Leftmost position=1
```

In the display program change the semicolon in the PRINT statement to a comma. This causes numbers to be displayed in four columns on a 40-column display. At four numbers per line, the TO index will be 4*20=80. When you run this program, note that the first position in each field is reserved for the sign.

```
C=2001:FOR I=1 TO 80:?C,:NEXT:?"PHEW!"

 2001           2001           2001           2001
 2001           2001           2001           2001
 2001           2001           2001           2001
 2001           2001           2001           2001
 2001           2001           2001           2001
 2001           2001           2001           2001
 2001           2001           2001           2001
 2001           2001           2001           2001
 2001           2001           2001           2001
 2001           2001           2001           2001
 2001           2001           2001           2001
 2001           2001           2001           2001
 2001           2001           2001           2001
 2001           2001           2001           2001
 2001           2001           2001           2001
 2001           2001           2001           2001
 2001           2001           2001           2001
 2001           2001           2001           2001
 2001           2001           2001           2001
 2001           2001           2001           2001
PHEW!

READY.
▒
```

Using commas between PRINT statement variables is a convenient way to tab. Now change the value of C from 2001 to 44. Change the TO index from 134 to 200. Press RETURN and (surprise!) you will get the display shown below.

```
C=44:FOR I=1 TO 200:?C;:NEXT:?"PHEW!"
 440 442 44  440 442 44  440 442 44  440
 442 44  440 442 44  440 442 44  440 442
 44  440 442 44  440 442 44  440 442 44
 440 442 44  440 442 44  440 442 44  440
 442 44  440 442 44  440 442 44  440 442
 44  440 442 44  440 442 44  440 442 44
 440 442 44  440 442 44  440 442 44  440
 442 44  440 442 44  440 442 44  440 442
 44  440 442 44  440 442 44  440 442 44
 440 442 44  440 442 44  440 442 44  440
 442 44  440 442 44  440 442 44  440 442
 44  440 442 44  440 442 44  440 442 44
 440 442 44  440 442 44  440 442 44  440
 442 44  440 442 44  440 442 44  440 442
 44  440 442 44  440 442 44  440 442 44
 440 442 44  440 442 44  440 442 44  440
 442 44  440 442 44  440 442 44  440 442
 44  440 442 44  440 442 44  440 442 44
 440 442 44  440 442 44  440 442 44  440
 442 44  440 442 44  440 442 44  440 442
PHEW!HEW!

READY.
▒
```

Some of the digits from the previous 2001 display were not blanked out. **CBM BASIC uses a skip (cursor right) character, not blanks, between fields. When you display over existing data, characters between fields — or characters in tabbed format — are not erased.** Also note that there is a remaining "HEW!"; CBM BASIC displays the "PHEW!" but leaves the remaining positions of the line just as they were; it does not

blank the rest of the line. **This can be a great advantage when you are adding to data already on the screen, and you should bear this capability in mind.** For the display program line, however, it is leaving extraneous characters in the display.

To remove extraneous characters from the display, you can have the program clear the screen before beginning a new display. To do this, insert a PRINT CLEAR SCREEN statement ahead of the FOR-NEXT loop:

```
C=44:?"■":FOR I=1 TO 200:?C$;:NEXT:?"PHEW!"
```

Clear Screen (shift of CLR/HOME key)

Now when you press RETURN, you will see the screen blank and the numbers displayed on the *second* line.

To begin displaying on the *first* line, insert a semicolon after the PRINT CLEAR SCREEN statement.

```
C$="■":?"■";:FOR I=1T0840:?C$;:NEXT:?"PHEW!"
```

With the extra line of forty characters, the program can display 840 characters without scrolling any off the top.

Commas also work when printing strings. As an example, enter the following immediate mode program to display twenty lines of tabbed character data:

```
A$="HUP!":B$="TWO!":C$="THREE!":D$="FOUR!"
FOR I=1 TO 20:?A$,B$,C$,D$:NEXT:?"PHEW!"
```

CURSOR MOVEMENT

In Chapter 3 we discussed the screen editing capabilities provided by the cursor control keys: CLEAR SCREEN/HOME, CURSOR UP/DOWN, CURSOR LEFT/RIGHT, INSERT/DELETE, and RETURN.

The CLEAR SCREEN/HOME, CURSOR UP/DOWN, CURSOR LEFT/RIGHT, and REVERSE keys can be included within PRINT statement strings. The INSERT/DELETE key and the RETURN key cannot be used within a PRINT statement.

Cursor control keys are interpreted as characters within a string until the PRINT statement is executed. Consider the PRINT statement:

```
100 PRINT"*■■*"
```

Quotation set #2: change program mode to immediate mode
Programmed representation of cursor right
Quotation set #1: change immediate mode to program mode

When this PRINT statement is executed, you can see the cursor has moved right by the placement of the asterisks:

```
RUN

* *
```

To practice simple programmed cursor movement, type in the following program:

```
10 PRINT"<CLEAR SCREEN>";
20 PRINT"<CURSOR↓>*<CURSOR↓>*<CURSOR↓>*<REVERSE><CURSOR↑>*
   <CURSOR↑>*<CURSOR↑>*"
30 PRINT"<CURSOR→><CURSOR→><CURSOR→><CURSOR→>";
```

The program should look like this on your screen:

```
10 PRINT"□";
20 PRINT"■■■■■□■□■";
30 PRINT"■■■■■"
40 END
```

Upon execution, the output should appear as follows:

```
      ■
*    ■
 *  ■
  *
```

This may or may not have been what you expected. If you expected the character sequence:

```
20 PRINT "■■■■■          "
```

to display the asterisks in a vertical line:

```
*

*

*
```

or if you expected the character sequence:

```
20 PRINT "          □■□■□■"
```

to display three asterisks back up over the original three:

```
\         ■

          ■

          ■
```

you forgot about the **automatic right movement of the cursor following every keystroke.** The programmed cursor control causes the CBM computer to move the cursor directly up or directly down, but the asterisks will be displayed in a diagonal line due to the cursor's automatic advance. Each time a character is displayed, the cursor is automatically advanced one space to the right. This prevents the last character from being overwritten. The following diagram shows the cursor movement of the previous program:

To display a vertical line you must compensate for the advance by moving the cursor back one space to the left before moving it up one space or down one space. For example, the following program statement displays a vertical descending line of three asterisks followed by a vertical line of three ascending, adjacent asterisks:

```
20 PRINT "<CURSOR↓>*<CURSOR←><CURSOR↓>*<CURSOR←><CURSOR↓>*
   <REVERSE>*<CURSOR←><CURSOR↑>*<CURSOR←><CURSOR↑>*";
```

This will be displayed as follows:

```
20 PRINT"■■■■■□■□■";
```

If you attempt to program the INSERT/DELETE and the RETURN keys, you will encounter some surprising results.

The INSERT key is programmable. When you press the INSERT key between a set of quotes, a reverse capital T displays. Of course the CBM will not appear to insert a space if the entire line the cursor is on is blank.

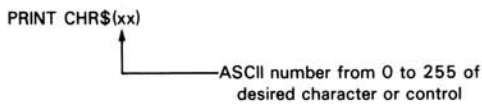
The DELETE key remains in immediate mode. Trying to program the DELETE key in a PRINT statement will merely erase the previous character, unless the DELETE key occurs within a sequence of inserted characters. The DELETE key is programmable following an insert, but do not use it in this fashion. It will simply get you into trouble. There are simpler ways of achieving the same objective in a program.

The RETURN character in a PRINT statement will immediately move the cursor out of the statement and to the next line.

CHR$ FUNCTION: PROGRAMMING CHARACTERS IN ASCII

If you cannot press a key to include a character within a text string, you can still select the character by using its ASCII value.

The CHR$ function translates an ASCII code number into its character equivalent. The format of the CHR$ function is:



To obtain the correct ASCII code for the desired character, refer to Appendix A. Scan the columns until you find the desired character or cursor control, then note the corresponding ASCII code number. Insert this number between the two parentheses of the CHR$ function. For example, to create the symbol "$" from its ASCII code, find the ASCII code for "$." "$" has two ASCII values: 36 and 100. Which value should you use? Either number works just as well. But for good programming technique, once you select one number over the other, use that number consistently throughout the program. We will use 36 and insert it into the CHR$ function as follows:

```
PRINT CHR$(36)
```

Try displaying this character in immediate mode:

```
PRINT CHR$(36)
$
```

Now, try displaying ASCII code 100:

```
PRINT CHR$(100)
$
```

The result is the same. Experiment in immediate mode using any ASCII code from 0 to 255.

You can use the CHR$ function in a PRINT statement as follows:

```
10 PRINT CHR$(36);CHR$(42);CHR$(166)

RUN
$*▒
```

The CHR$ function lets you include otherwise unavailable characters such as RETURN, INSERT/DELETE, and the quote character (") among a PRINT statement's parameters. You may also use the CHR$ function to do comparison checking for cursor controls such as RETURN and INSERT/DELETE. Suppose a program must check characters input at the keyboard, looking for a RETURN key. You could check for a RETURN (which has an ASCII code of 13) as follows:

```
10 GET X$:IF X$<>CHR$(13) THEN 10
```

This test would be impossible if you tried to put RETURN between quotation marks:

```
20 IF X$<>" RETURN "THEN 10
```

Impossible

This is impossible because when you depress the RETURN key following a set of quotes, it automatically moves the cursor to the next line:

```
20 IF X$<>"
```

Press RETURN key

Cursor Controls (CBM 8000)

The screen editor release 4.1, available on the CBM 8000, has two new key functions and some new edit/control capabilities. The key functions are provided by the TAB and ESCAPE keys. The edit/control capabilities include a programmable bell, line insert and delete, screen erase, graphic/text switching, and scrolling within a programmable screen window.

The TAB Key and Tabbing Function. **The TAB key operates much like a typewriter TAB key. The tabbing capabilities of the TAB key are equivalent to the TAB function. Up to 80 TABs may be set per line.** To set a TAB in immediate mode, move the cursor to the desired screen column, then press the TAB and SHIFT keys simultaneously. When all tabs have been set, press the RETURN key:

Press shifted TAB key

You can **program the TAB SET using a PRINT statement.** The text string to be printed must move the cursor to the required column, then execute a TAB SET. The TAB SET character is generated by pressing the shifted TAB key. This may be illustrated as follows:

```
PRINT"
```

TAB SET

CURSOR RIGHT

A reverse upper-case I is displayed for the TAB SET.

The TAB SET is represented by ASCII value 137, therefore TAB SET can be programmed using the CHR$ function:

```
PRINT"     ";CHR$(137)
```

The TAB key advances the cursor to the next tabbed column on the screen. To tab the cursor in immediate mode, simply press the TAB key. If TAB is pressed beyond the last tab position on the screen, the cursor jumps to the end of the display line.

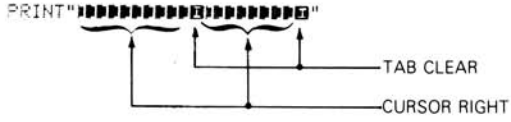
When included in a PRINT statement, the tab will occur at the point where the TAB character is encountered. Here is an immediate mode example:

```
PRINT" MY□PET BITES"
          ↑
          └──────────Programmed TAB
MY PET BITES
```

TAB CLEAR clears a TAB SET position. In immediate mode move the cursor to the column whose tab set is to be cleared, then press the TAB and SHIFT keys simultaneously. Following the last TAB CLEAR, press the RETURN key.

TAB CLEAR and TAB SET are both generated by the shifted TAB key. Therefore **if you try to clear a tab in a column where none was set, you will set a tab instead.**

Tabs are cleared in program mode using a PRINT statement that moves the cursor to the required column, then executes a TAB CLEAR character:



TAB CLEAR, like TAB SET, is displayed as a reverse upper-case I character.

TAB CLEAR can be programmed using the CHR$ function as follows:

```
PRINT"■■■■■■";CHR$(137)
```

CHR$(137) represents the TAB SET and TAB CLEAR characters.

Escape. The ESCAPE key on the CBM 2001/B business keyboard generates an ASCII code, but has no editing capabilities. **On the CBM 8000 keyboard the ESCAPE key has two functions: pressed in immediate mode it cancels an insert, reverse, or text entry condition. ESCAPE also allows certain character strings to be interpreted as screen editing control functions.**

ESCAPE can be included in a PRINT statement by using the CHR$ function. Enter:

```
PRINT CHR$(27)
```

Control Functions (CBM 8000)

Control functions summarized below are available only on the CBM 8000 computers with the 80-column screen. These functions are defined in detail in Chapter 8. Some examples of their use are given later in this chapter.

All of these control functions are desgined to improve displays and data entry; although they can be used in immediate mode, they should not be used to edit programs. Many of these functions modify the display without simultaneously changing memory content.

To use one of these functions, its character must appear in a PRINT statement's parameter list. The function character can be specified within a text string using a control character, or it may be specified outside of a text string using a CHR$ function. The control character is generated by pressing the ESCAPE key, then the REVERSE key, then the appropriate unshifted letter key.

Bell. The Bell function works only on a CBM 8000 computer that is equipped with a bell. The bell will ring automatically on power-up, and whenever the cursor moves through column 75. If the screen window has been narrowed (using the scrolling window function) the bell will sound as the cursor passes through the fifth column from the right edge of the window. The bell is also sounded by a Control-g character, or a CHR$(7) function in a PRINT statement.

Delete Line and Insert Line. These functions delete or insert a display line. The Delete Line function deletes the line on which the cursor is located; all lower lines on the display are scrolled up one line position. The Insert Line function inserts a line at the cursor screen location, scrolling all lower lines down; the bottom line is scrolled off the screen. Neither the Delete nor the Insert Line function modifies computer memory; only the display changes. The Delete Line function is generated by a Control-u character or the CHR$(21) function in a PRINT statement parameter list. The Insert Line function is generated by a Control-M character or a CHR$(149) function in the PRINT statement parameter list.

Erase Begin and Erase End. These functions erase part of the line on which the cursor is currently positioned. The Erase Begin function erases all text to the left of the cursor; the Erase End function erases all text to the right of the cursor. Neither function moves remaining text. Neither function modifies memory. The Erase Begin function is generated by a Control-V character or a CHR$(150) function occurring in a PRINT statement parameter list. The Erase End function is generated by a Control-v character or a CHR$(22) function appearing in a PRINT statement parameter list.

Graphic or Text. The Graphic function selects graphic characters from the standard character set, while text characters select upper- and lower-case letters. Also, spaces between graphic characters are eliminated in order to improve the quality of graphics. The graphic function is selected by a Control-N character or a CHR$(142) function appearing in a PRINT statement parameter list.

The Text function is the inverse of the Graphic function. The Text function selects the alternate character set for graphic characters, while text characters continue to select upper- and lower-case letters. The Text function is selected by a Control-n character or the CHR$(14) function appearing in a PRINT statement parameter list.

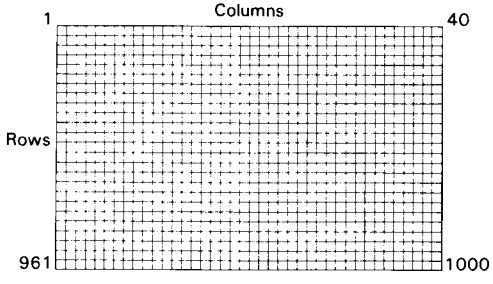
Screen Window Functions. There are four functions which allow a window to be defined in the CBM 8000 display, with text scrolled up or down within the defined window. The **Set Top function** takes the current cursor location as the top left-hand corner of the display window; the **Set Bottom function** takes the current cursor location as representing the bottom right-hand corner of the window. This window can be canceled at any time by pressing the HOME key twice, or by executing a PRINT statement with two contiguous HOME characters in its parameter list. Set Top is selected by the CHR$(15) function and Set Bottom is set by the CHR$(143) function; these CHR$ functions should appear in a PRINT statement parameter list following cursor move characters that correctly position the cursor to define the top left and right bottom corners of the window.

The **Scroll Up function** moves text up one line within a window defined by the Set Top and the Set Bottom functions. A blank line is inserted at the bottom of the window. The **Scroll Down function,** likewise, moves text down one line within the window, inserting a blank line at the top of the window. Scroll Up is selected by a Control-q character or the CHR$(25) function. Scroll Down is selected by a Control-Q or the CHR$(153) function. The control character or CHR$ function must appear in a PRINT statement parameter list.

POKE to the Screen

You can use a POKE statement to display any character anywhere on the screen. Simply **POKE the character value into the correct screen location in memory.**

The CBM computer screen is like a grid of 1000 (or 2000) squares, organized as 25 rows and 40 (or 80 columns). A 40-column display may be illustrated as follows:



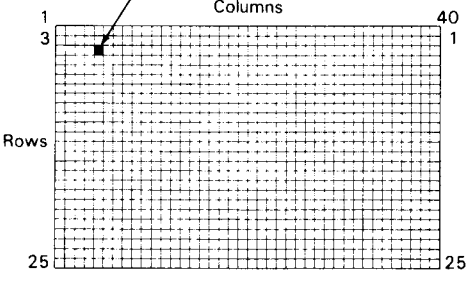
One character may be displayed in each square. **Every screen location is assigned an address and space in memory.** Memory screen space begins at address 32768 for square 1 (row 1, column 1) and ends at address 33767 for square 1000 (row 25, column 40), or at address 34767 for square 2000 (row 25, column 80). Memory address 32768 is screen location (1,1), address 32769 is screen location (1,2), etc. Figure 5-2 shows the correlation between screen locations and their corresponding memory spaces and addresses.

To find the screen address in memory for any screen location, use the following equations:

40 Column Screen	80 Column Screen
32768+(column-1)+(40*(row-1))	32768+(column-1)+(80*(row-1))

Enter the column and row numbers of any screen position into the equation to find its memory address. To demonstrate, enter the values 5 and 3 to find the memory address for the screen location at column 5, row 3:

```
=32768+(COL-1)+(40*(ROW-1))
=32768+(5-1)+(40*-1))
=32768+4+(40*2)
=32768+4+80
=32852
```



The memory address for screen location (5,3) is 32852.

This equation makes it possible to POKE characters to the screen without knowing any more than the column and row number of the location to be POKEd. Recall the format of the POKE statement:

```
POKE A,X
```

where:

A is the screen address.

X is the character or variable to be POKEd into A.

Replace A with the screen equation and the computer will calculate the screen address for you:

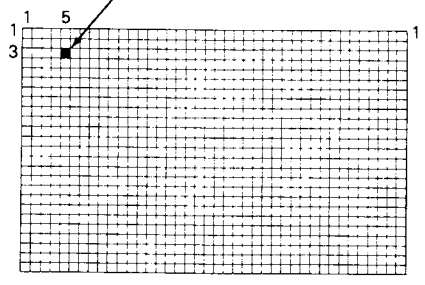
```
POKE 32768+(COL-)+(40•ROW-1),X
POKE          A          ,X
```

For instance, if COL (C) and ROW (R) is input as 5,3, and X is input as ♠, then a spade will be POKEd at screen location (3,5), address 32852.

Try entering and executing this program:

```
10 INPUT C,R,X
20 POKE 32768+(C-1)+(40*(R-1)),X
```

```
=32768+(COL-1)+(40•(ROW-1)),X
=32768+(5-1)+(40•(ROW-1)),X
=32768+4+(40•2),X
=32768+4+80,X
=32852,X
```



X is entered as a number in the range 0 to 255. The ASCII character corresponding to the entered number will be displayed.

Variables may be used in POKE statements, but the variable must evaluate to a number within the allowed limits:

```
POKE 32768+A,X
```

where: A is a number between 0 and 999 inclusive (32768+999=33767) for a 40 column screen

```
POKE A, X
```

where: A is a number between 32768 and 33767 inclusive for a 40 column screen

Using a variable to represent the screen address is wise when POKEing to a repeating sequence of screen spaces. For example, the program below POKEs the value of X ten spaces apart across the screen:

```
10 A=32768
20 POKE A,X
30 A=A+10
40 IF A<=33767 GOTO 20
```

DATA ENTRY (INPUT)

Data entry should be programmed in functional units.

A mailing list program, for example, requires names and addresses to be entered as data. You should treat each name and address as a single functional unit. In other words, your program should ask for the name and address, allowing the operator to enter all of this information and then change any part of it; when the operator is satisfied that the name and address are correct, the program should process the entire name and address as a single functional unit. Then the program should ask for the next name and address.

It is bad programming practice to break up data input into its smallest parts. In the case of a mailing list program it would be bad programming practice to ask for the name, process this data as soon as it has been entered, then ask for each line of the address, treating each piece of the name and address as a separate and distinct functional unit.

The goal of any data entry program should be to make it easy for an operator to spot errors and to give the operator as many chances as possible to fix errors.

Suppose a program requires a long list of short, identical data items to be input. Such a list may consist of names, social security numbers, or perhaps dates. It is a good idea to write a program which accepts such input in blocks. For example, if names must be entered, the program might allow the operator to enter as many names as will fit in one vertical column, so that any entry can be corrected while it is still being displayed. The program would accept and process names as they scroll off the top of the screen. The alternative would be to write an input program that accepts and processes one name at a time. But this program would reduce an operator's chances of spotting and correcting mistakes.

There is one set of circumstances when entering data in blocks is not the best way to go, and this set of circumstances is a surprising one: it occurs when a very large amount of data must be entered by keyboard operators. For example, suppose a keyboard operator must enter hundreds of names and addresses a day. Experience has shown that the highest volume of accurate data entry can be achieved by having the keyboard operator ignore all errors on first entry. The data entry program should not allow for the correction of any errors, even if the errors are detected as data is being entered. Operators should be trained to ignore errors and carry on entering data as fast as possible. Such data should be entered twice, preferably by different operators. The data entry should be compared. The chances of both operators making the same error are so small that you can count on all errors being flagged as differences between the two sets of data entry. A subsequent program should allow incorrect data to be corrected.

Interactive Data Input

To demonstrate the value of good, interactive data input we will begin with a very simple example. Starting with an early version of program BLANKET we will discuss, step by step, the changes that improve data entry, thereby making the program easier to use.

Start with the program listed below; we will finish with the program as it appears in Figure 5-1.

```
100 C$="A"
110 PRINT "□"
120 FOR I=1 TO 840
130 PRINT C$;
140 NEXT
150 PRINT "PHEW!"
```

The program above will display 800 A's followed by the exclamation PHEW!

Suppose we want to display X's instead of A's.
First eliminate the assignment statement in the program. To delete a program statement, type the line number followed immediately by a RETURN.

```
LIST 100

 100 C$="A"

READY.
100◄──────────────Type line number, then key RETURN.
LIST

 110 PRINT "□";
 120 FOR I=1 TO 840
 130 PRINT C$;
 140 NEXT
 150 PRINT "PHEW!"
READY.
▒
```

Line 100 is no longer in the program. Type in the statement C$="X" in immediate mode, then run the program.

Before RETURN	After RETURN
C$="X"	PHEW!
READY.	READY.
RUN▒	▒

The screen blanks and the word PHEW! is printed, but the X's are not printed. Obviously the value of C$ is not being transmitted to the program.

RUN clears all variables to 0 and all strings to null before beginning execution of a program. So C$ was set to null, and a null character was printed in the program loop (a "null" is "nothing": it does not print nor does it move the cursor).

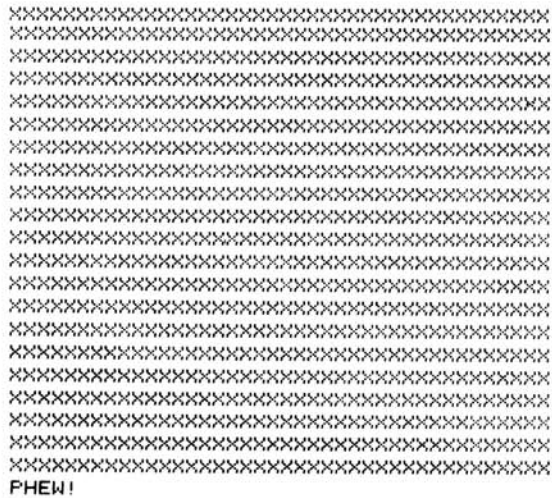
Is there a way to transmit the value of C$, entered in immediate mode, to the program? Instead of using RUN, which initializes variables, use GOTO 110 (110 being the line number of the first line of the program). This does not change any variable values.

Before RETURN

```
C$="X"

READY.
GOTO 110
```

After RETURN



Now the procedure for running the program is as follows:

1. In immediate mode enter the assignment C$="y" where y is any display character.
2. Enter the immediate statement GOTO 110.

There are only two steps in running the program, but the procedure is awkward. You must type in a line (the assignment statement), and if you enter RUN instead of GOTO, you must start all over. **But the program could fetch the display character while it is running,** using the GET statement. Type in the following line:

```
100 GET C$:IF C$="" GOTO 100
```

List the program and make sure you entered the line correctly. Then run the program. The screen blanks and the cursor disappears. Press any data key. The character you enter is displayed 800 times. Run the program again. Press any data key. The display appears with the new character.

Here is the new procedure for running program BLANKET:

1. Enter the RUN command.
2. Press any key.

This is a real improvement over the original program. However, it is a little disconcerting to have the screen completely blank out while it waits for you to press a key. Add a prompt line to the beginning of the program, asking for key to be pressed. Type in the line:

```
90 ?"HIT A KEY"
```

List the program and check the new line for errors.

Now the program gives operating instructions. Run the program several times to display different characters and note how much easier the program is to use.

There is one important modification left to make. If you want **to run the program more than once,** go back to the beginning of the program instead of ending it. Then you won't have to type in RUN to reexecute the program. Add the following line:

```
160 GOTO 90
```

Again, list the program and check the new line. It should look like this:

```
LIST

 90 PRINT "HIT A KEY"
 100 GET C$: IF C$="" GOTO 100
 110 PRINT "□";
 120 FOR I=1 TO 840
 130 PRINT C$;
 140 NEXT
 150 PRINT "PHEW!"
 160 GOTO 90
READY.
■
```

Now it is even easier to use the program. Enter RUN and follow directions.

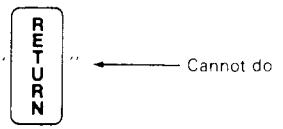
Of course, you have to use the STOP key to exit from the program. This can be eliminated by programming one particular key **to terminate program execution.** For example, the RETURN key could be programmed to terminate execution.

Let us see how this is done.

All data keys and cursor control keys can be checked as string characters. For example, the following statements check for a 'Y' character:

```
100 GET C$: IF C$="" GOTO 100
105 IF C$="Y" GOTO 200
```

RETURN presents a special problem. You cannot reference RETURN as a string literal:



This is because any time RETURN is pressed, CBM BASIC stores the program line in memory and goes to the beginning of the next line. You can, however, use the CHR$ function to check for a RETURN key entry. CHR$ allows you to assign an ASCII code value to a string variable and treat it as a string. The ASCII code value for a RETURN is 13.

Before programming to check for a carriage return, consider what must be done if there is one. The last line of the program branches back to the beginning of the program. To terminate program execution, you need to branch beyond the last line. Add the following line:

```
170 END
```

Now add the check for RETURN for program termination at line 105:

```
105 IF C$=CHR$(13) GOTO 170
```

Note that we could have written, in place of line 170 and line 105:

```
105 IF C$=CHR$(13) THEN END ←— Option
```

If you choose this option, it is generally good programming practice to have the program termination point at the physical end of the program. It is more difficult to find termination points embedded in the program.

Without the READY message being printed each time, there are two additional lines available on the screen. This allows 80 more characters (at 40 characters per line) to be printed. Change the number of characters on line 120 from 840 to 840+80=920. Line 120 will read:

```
120 FOR I=1 TO 920
```

When you run the program, you will find that it is scrolling up one line, leaving a blank line at the bottom of the screen. This is because CBM BASIC executes a RETURN after displaying HIT A KEY; it does this to select a new line in preparation for the next display. We can demonstrate this by making the cursor blink.

Normally you cannot see the cursor because its blinking is inhibited before a program is run. However, you can make the cursor blink by adding the following statement to the beginning of the program:

```
30 POKE 548,0◄——— Enable cursor
```

This is a system location that is discussed further in Chapter 7. Run the program with this line added and you will see the cursor blinking at the bottom line.

```
.
.
PHEW!
HIT A KEY
```

This program does not really need the cursor, so delete line 80.

To prevent the blank line at the bottom of the screen, add a semicolon to the PRINT statement in line 90. We should also add a prompt that RETURN is used to exit from the program. To incorporate these changes, line 90 should now be edited as follows:

```
90 ?"HIT A KEY OR <R> TO END";
```

As a final task, you might read over the program and add remarks. Comment on how the number 920 was devised; you can optionally put the remark on the same line, using a colon to separate statements:

```
120 FOR I=1 TO 920    :REM 920/40=23 LINES
```

Add a reminder that the screen is cleared; optionally align the remarks:

```
110 PRINT"□";          :REM CLEAR SCREEN
```

Finally, add a few lines at the beginning of the program to describe it. The final program BLANKET is shown in Figure 5-1. Save it on tape or diskette.

Prompting Messages

Any program that requires data entry should prompt the operator by asking questions. Questions are usually displayed on a single line and demand a simple response such as "yes," "no," a word, or a number. For example the following message might be displayed:

```
DO YOU WANT TO MAKE ANY CHANGES?
```

An operator must respond to this message by entering the word YES or the word NO. Frequently just the letter Y or N suffices. Another common example may give the operator a number of options. The message:

```
WHICH ENTRY DO YOU WISH TO CHANGE?
```

may allow the operator to enter a number which identifies an entry.

Programs that control this type of dialogue should be written as stand-alone subroutines which do not depend on knowledge of the calling program. This has three implications:

1. You cannot assume that the row on which the message will be displayed is blank. If the row is not blank, then the message will overwrite whatever was previously there; but worse, the remainder of the line, beyond the message, will be interpreted as part of the response. This is ugly from the operator's viewpoint, but it can also be troublesome. Depending on how your program is written, remaining characters beyond the message may be interpreted as part of the data input.
2. The subroutine must receive parameters from the calling program. For example, if a message asks the operator to enter a number, then the calling program should pass the minimum and maximum allowed numbers to the subroutine as parameters.
3. The subroutine must return the operator's response to the calling program. This variable may be a character (e.g., Y or N), it may be a word (e.g., yes or no), or it might be a number.

Subroutine logic cannot deduce on which screen row the message is to appear. It is therefore fair to demand that the calling program position the cursor on the correct row. You can clear the selected row and position the cursor at column 0 of the row using the following statements:

```
2000 REM CLEAR THE ROW ON WHICH THE CURSOR IS CURRENTLY POSITIONED
2010 PRINTCHR$(13);"□";:REM MOVE CURSOR TO COLUMN 0
2020 FOR I=1 TO 39:PRINT" ";:NEXT
2030 PRINTCHR$(13);"□";
2040 STOP
```

For an 80-column screen the statements on line 2020 should write 79 blanks, rather than 39 blanks as illustrated.

Enter this program into your computer; position the cursor on a blank line between two lines of text, then type RUN <CR> to execute the program. If all the text scrolls off the top of the screen then you forgot the semicolon that must terminate the PRINT statement on line 2020.

Frequently the statements illustrated above will be called as a subroutine, in which case a RETURN statement must occur on line 2040.

Alternatively you can use the **CBM 8000 Erase Begin and Erase End** functions:

```
2000 REM CLEAR THE ROW ON WHICH THE CURSOR IS CURRENTLY POSITIONED
2010 PRINTCHR$(150);CHR$(22);CHR$(13);"□";
2030 STOP
```

The routine collapses to a single statement. Calling this single statement as a subroutine would be pointless.

Now look at the subroutine needed to ask a question that requires a reply of Y for yes, or N for no. We will use a PRINT statement to ask the question, followed by a GET to receive a one-character response. Clear the row on which the question is to be asked by calling the clear row subroutine. Here is the program and the called subroutine:

```
2000 REM CLEAR THE ROW ON WHICH THE CURSOR IS CURRENTLY POSITIONED
2010 PRINTCHR$(13);"□";:REM MOVE CURSOR TO COLUMN 0
2020 FOR I=1 TO 39:PRINT" ";:NEXT
2030 PRINTCHR$(13);"□";
2040 RETURN
3000 REM ASK A QUESTION AND RETURN A RESPONSE OF Y OR N IN YN$
3010 GOSUB 2000
3020 PRINT"DO YOU WANT TO MAKE ANY CHANGES? ";
3030 GET YN$:IF YN$<>"N" AND YN$<>"Y" THEN 3030
3040 PRINTYN$;
3050 RETURN
```

You can use the program illustrated above to ask any question that requires a "yes" or "no" response. The message to be displayed, whatever it may be, must occur in the PRINT statement on line 3020.

Next consider dialogue which allows an operator to enter a number. We will assume that the subroutine receives the smallest number in integer variable LO% and the largest number in integer variable HI%. The subroutine will return the entered number in NM%. Here is the necessary program:

```
2000 REM CLEAR THE ROW ON WHICH THE CURSOR IS CURRENTLY POSITIONED
2010 PRINTCHR$(13);"□";:REM MOVE CURSOR TO COLUMN 0
2020 FOR I=1 TO 39:PRINT" ";:NEXT
2030 PRINTCHR$(13);"□";
2040 RETURN
3000 REM ASK FOR A NUMERIC SELECTION
3001 REM RETURN SELECTION IN NM%
3002 REM NM% MUST BE LESS THEN HI% AND MORE THAN LO%
3003 REM CALLING PROGRAM MUST SET HI% AND LO%
3010 GOSUB 2000
3020 PRINT"WHICH DO YOU WANT TO CHANGE? ";
3030 GET NM$:IF NM$="" THEN 3030
3040 NM%=VAL(NM$)
3050 IF NM%<LO% OR NM%>HI% THEN 3030
3060 PRINTNM$;
3070 RETURN
```

Write a short program that sets values for HI% and LO%, then goes to subroutine 3000. Add the subroutine illustrated above and run it. A CBM 8000 version of this program will replace the GOSUB statement on line 3010 with the PRINT statement on line 2010 of the previous program.

Can you change the subroutine so that it accepts two-digit inputs? Try to write this modified program for yourself. If you cannot do it, then wait until the next section, where you will find the necessary subroutine in the program which controls input of a date.

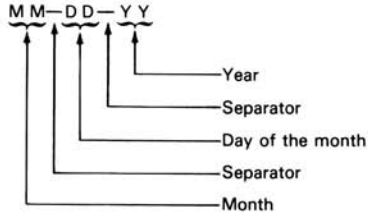
There is another simple modification you can make to both of the dialogues we have described; the message printed on line 3020 in both programs could be supplied by the calling program via a string variable. This would make the subroutines more general purpose. Can you rewrite the programs to accept a message provided by the calling program?

Entering a Valid Date

Most programs at some point need relatively simple data input: more than a simple yes or no, but less than a full screen display. Consider a date.

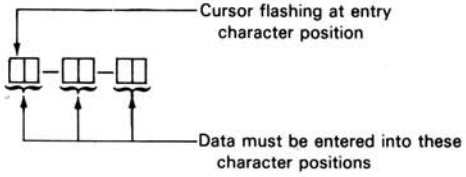
You must take more care with such simple data entry than might at first appear necessary. In all probability the date will be just one item in a data entry sequence. By carefully designing data entry for each small item, you can avoid having to restart a long data entry sequence whenever the operator makes an error in a single entry.

We will assume that the date is to be entered as follows:



Depending on your personal preferences, the dash separating two digit entries might be a slash, or any other visually pleasing character. In Europe the day of the month precedes the month.

Program data entry so that it is pleasing to the operator's eye. The operator should be able to see immediately where data is to be entered, what type of data is required, and how far the data entry process has proceeded. A good way of showing where data is to be entered is to reverse the data entry field. For example, the program that asks for a date to be entered might create the following reverse field display:



You can create such a display very simply using the following PRINT statement:

```
10 PRINT "<Clear><Cursor↓><Cursor↓>";TAB(20);"<Reverse>␣␣
   <Reverse off>-<Reverse>␣␣<Reverse off>-<Reverse>
   ␣␣<Reverse off>";CHR$(13);"<Cursor↑>";TAB(20);
```

␣ represents a space code.

The PRINT statement above includes cursor controls that position the date entry beginning at column 21 on row 3. Also, the PRINT statement clears the screen so that residual garbage display does not surround the request for a date. After displaying the date entry field, the PRINT statement moves the cursor back to the first character position of the first entry field by executing a carriage return, CURSOR UP and tab.

Try using an INPUT statement to receive entry of the month. This could be done as follows:

```
20 INPUT M$;
```

Enter statements on lines 10 and 20, as illustrated above, and execute them. The INPUT statement will not work. Apart from the fact that a question mark displaces the first reverse field character, the INPUT statement picks up the rest of the line following the question mark. Unless you overwrite the entire data entry display — and that

requires entering a very large number — you will get a RE-DO FROM START message each time you press the RETURN key.

This is an occasion to use the GET statement:

```
10 PRINT"■■■";TAB(20);"■ ■■ ■■ ■■";CHR$(13);"■";TAB(20);
20 GET C$:IF C$="" THEN 20
30 PRINTC$;:MM$=C$
40 GET C$:IF C$="" THEN 40
50 PRINTC$;:MM$=MM$+C$
60 STOP
```

These statements accept a two-digit input. The input is displayed in the first reverse field of the date. The two-digit input needs no carriage return or other terminating character; the program automatically terminates the data entry after two characters have been entered.

Three two-digit entries are needed: one each for the month, the day, and the year. Rather than repeating statements on lines 20 through 50, we will put these statements into a subroutine and branch to it three times, as follows:

```
10 PRINT"■■■";TAB(20);"■ ■■ ■■ ■■";CHR$(13);"■";TAB(20);
20 GOSUB 1000:MM$=TC$:PRINTTAB(23)
30 GOSUB 1000:DD$=TC$:PRINTTAB(26)
40 GOSUB 1000:YY$=TC$
50 STOP
1000 REM TWO CHARACTER INPUT SUBROUTINE
1010 GET C$:IF C$="" THEN 1010
1020 PRINTC$;
1030 GET CC$:IF CC$="" THEN 1030
1040 PRINTCC$;
1050 TC$=C$+CC$
1060 RETURN
```

If you have a CBM 8000 computer, try rewriting the program above to use the TAB SET and TAB functions provided by the Editor release 4.1.

A CBM 8000 version of this program is much simpler because you can use the **CBM 8000 Erase End** function, as follows:

```
3000 REM ASK A QUESTION AND RETURN A RESPONSE OF Y OR N IN YN$
3005 REM CBM 8000 VERSION
3010 PRINTCHR$(150);CHR$(22);CHR$(13);"■";
3020 PRINT"DO YOU WANT TO MAKE ANY CHANGES? ";
3030 GET YN$:IF YN$<>"N" AND YN$<>"Y" THEN 3030
3040 PRINTYN$;
3050 RETURN
```

The variables MM$, DD$, and YY$ hold the month, day, and year entries, respectively. Each entry is held as a two-character string. As described earlier in this chapter, you should empty the ten-character input buffer before accepting the first input, otherwise any prior characters in the input buffer will be read by the first GET statement in the two-character input subroutine. You only need to empty the buffer once, before the first GET statement.

There are two ways in which we can help the operator recover from errors while entering a date.

1. The program can automatically test for valid month, day, and year entries.
2. The operator can be given a means of restarting the data entry.

The program can check that the month lies between 01 and 12. The program will not bother with leap year, but otherwise it will check for the maximum number of days in the specified month. Any year from 00 through 99 will be allowed. Any invalid entry will cause the entire date entry sequence to restart.

If the operator presses the RETURN key, then the entire date entry sequence restarts.

Our final date entry program now appears as follows:

```
5 REM ROUTINE TO ACCEPT AND VERIFY A DATE
10 PRINT"■■■";TAB(20);"■ ■■ ■■ ■■";CHR$(13);"■";TAB(20);
50 GOSUB 1000:REM GET MONTH
60 IF C$=CHR$(13) OR CC$=CHR$(13) THEN 10
70 DT$=TC$:PRINTTAB(23)
80 REM CHECK FOR VALID MONTH
90 M%=VAL(TC$)
100 IF M%<1 OR M%>12 THEN 10
110 REM GET NUMBER OF DAYS IN MONTH
120 D%=31
130 IF M%=2 THEN D%=28
140 IF M%=4 OR M%=6 OR M%=9 OR M%=11 THEN D%=30
150 GOSUB 1000:REM GET DAY
160 IF C$=CHR$(13) OR CC$=CHR$(13) THEN 10
170 DT$=DT$+"-"+TC$:PRINT TAB(26)
190 REM CHECK FOR VALID DAY
200 IF VAL(TC$)<1 OR VAL(TC$)>D% THEN 10
210 GOSUB 1000:REM GET YEAR
220 DT$=DT$+"-"+TC$
230 IF C$=CHR$(13) OR CC$=CHR$(13) THEN 10
240 REM CHECK FOR VALID YEAR
260 IF VAL(TC$)<0 OR VAL(TC$)>99 THEN 10
270 STOP
1000 REM TWO CHARACTER INPUT SUBROUTINE
1005 FOR I=1 TO 10:GET C$:NEXT:REM CLEAR OUT INPUT BUFFER
1010 GET C$:IF C$="" THEN 1010
1015 IF C$=CHR$(13) THEN 1050
1016 IF C$<"0" OR C$>"9" THEN 1010
1020 PRINTC$;
1030 GET CC$:IF CC$="" THEN 1030
1035 IF CC$=CHR$(13) THEN 1050
1036 IF CC$<"0" OR CC$>"9" THEN 1010
1040 PRINTCC$;
1050 TC$=C$+CC$
1060 RETURN
```

Notice that the date is built up in eight-character string DT$, as month, day, and year are entered.

These three checks are made on data as it is entered:

1. Is the character a RETURN?
2. If the character is not a RETURN, is it a valid digit?
3. Is the two-character combination a valid month for the first entry, a valid day for the second entry, or a valid year for the third entry?

The carriage return has been selected as an abort (restart) character. By replacing CHR$(13) on lines 60, 160, 230 and 1035 you can select any other abort character. When the operator presses the selected abort key the entire date entry sequence restarts. We must check for the abort character in the two-character input subroutine (at line 1035) since we want to abort after the first or second digit has been entered. The main program also checks for an abort character in order to branch back to the statement on line 10 and restart the entire date entry sequence. You could branch out of the two-character input subroutine directly to the statement on line 10 in the calling program, thereby eliminating the abort character test in the calling program. But this is a bad practice and we strongly discourage it. Every subroutine should be treated as a logical module, with specified entry point(s) and standard subroutine returns. Branching between the subroutine and the calling program is likely to be a source of programming errors. If you branch out of the subroutine and back to the calling program without

going through the return, you are laying yourself open to all kinds of subtle errors that you will not even understand until you are a very experienced programmer.

Program logic that tests for non-digit characters can reside entirely in the two-character input subroutine. We have chosen to ignore non-digit characters. Statements on lines 1016 and 1036 test for non-digit characters by performing comparisons between the ASCII value for the input character and the ASCII values for the allowed numeric digits.

Logic to check for valid month, day, and year must exist within the calling program since each of these two-character values have different allowed limits.

The statement on line 100 tests for a valid month.

Statements on lines 120, 130, and 140 compute the maximum allowed day for the detected month. The statement on line 200 checks for a valid day. The check for a valid year on line 260 is very simple.

Note that we generate an integer representation of the month on line 90, but we do not bother to generate integer representations of the day or the year. This is because the day and year are not used very often, but the month is used on lines 90 through 140. We will save both memory and execution time by using an integer representation of the month.

It takes more time to write a good data entry program that displays information in a pleasing manner and checks for valid data input, allowing the operator to restart at any time. Is the time worth spending? By all means yes. You will write a program once; an operator may have to run the program hundreds or thousands of times. Therefore you spend extra programming time once, in order to save operators hundreds or thousands of delays.

Forms Data Input

The best way of handling multi-item data entry is to display a form, and then fill in the form as data is entered. Consider a name and address. First display a form as follows:

```
ENTER NAME AND ADDRESS
 1  NAME:
 2 STREET:
 3   CITY:
 4  STATE:                                5 ZIP:
```

Notice that each entry has been assigned a number. The form displays the number in a reverse field.

The operator enters data sequentially, starting with item 1 and ending with item 5. The operator can then change any specific data entry.

The following program will clear the screen and display the initial form:

```
10 REM NAME AND ADDRESS DATA ENTRY
20 REM DISPLAY THE DATA ENTRY FORM
30 PRINT"■■ ENTER NAME AND ADDRESS"
40 PRINT" ■1■   NAME:"
50 PRINT" ■2■ STREET:"
60 PRINT" ■3■   CITY:"
70 PRINT" ■4■  STATE:";TAB(28);"■5■ ZIP:"
```

As each data item is entered, create a reverse field to identify the character field where data will appear as it is entered. Then as each character is entered, display it. The CURSOR LEFT key is used to restart data entry into the current field. The RETURN key ends data entry into the current field. The following instruction sequence provides us with necessary program logic:

```
80 REM GET 20 CHARACTER NAME
90 LN%=20
100 PRINT"■";TAB(10);
110 GOSUB 8000:NA$=CC$
120 REM GET 20 CHARACTER STREET
130 PRINTCHR$(13);TAB(10);
140 GOSUB 8000:SR$=CC$
150 REM GET 20 CHARACTER CITY
160 PRINTCHR$(13);TAB(10);
170 GOSUB 8000:CI$=CC$
180 REM GET 18 CHARACTER STATE
185 LN%=18
190 PRINTCHR$(13);TAB(10);
200 GOSUB 8000:ST$=CC$
210 REM GET 5 CHARACTER ZIP CODE
220 LN%=5
230 PRINTTAB(34);
240 GOSUB 8000:ZI$=CC$
250 STOP
8000 REM ENTER STRING DATA INTO A FIELD WITH LN% CHARACTERS
8010 REM THE CURSOR MUST BE IN THE FIRST CHARACTER POSITION OF THE FIELD
8020 REM THE RETURN KEY WILL END DATA ENTRY INTO THE FIELD
8030 REM THE ← KEY WILL RESTART DATA ENTRY INTO THE FIELD
8040 REM NO VALIDITY CHECKS ARE MADE ON ANY ENTERED DATA
8050 REM THE ENTERED STRING IS RETURNED IN STRING VARIABLE CC$
8060 ST%=POS(X):REM GET FIELD FIRST CHARACTER POSITION
8070 PRINT"■";:REM REVERSE ENTRY FIELD
8080 FOR I=1 TO LN%:PRINT" ";:NEXT
8090 PRINT"■";CHR$(13);"□";TAB(ST%);
8100 REM ENTER DATA AND DISPLAY AS ENTERED
8110 CC$="":J%=0
8120 FOR I=1 TO LN%
8125 J%=J%+1
8130 GET C$:IF C$="" THEN 8130
8140 IF C$="←" THEN PRINTCHR$(13);"□";TAB(ST%);:GOTO 8070
8150 IF C$=CHR$(13) THEN 8200
8160 PRINTC$;:CC$=CC$+C$
8170 NEXT
8190 REM FILL THE REST OF CC$ WITH BLANKS AND DISPLAY IT
8200 IF J%=LN% THEN 8300
8210 FOR I=J% TO LN%
8220 CC$=CC$+" "
8230 NEXT
8300 PRINTCHR$(13);"□";TAB(ST%);CC$;
8310 RETURN
```

Key in the entire program (statement 10 to statement 8310) and run it. Remember, if you still have statements 10 through 70 keyed into your computer you do not need to reenter these statements.

If your program does not run correctly, check your entry carefully. In particular, check for semicolons in PRINT statements.

When you run the program each of the five fields in turn will be highlighted by a reverse field. As you enter characters they will be displayed in the field. When you press the RETURN key the entire reverse field is replaced by the data you entered. Try pressing the CURSOR LEFT key to restart data entry.

Carefully read through the data entry subroutine, beginning at line 8060 and ending at line 8310. Before going any further you should clearly understand this program logic.

Note how easy it is for an operator to see what he or she is entering, and how simple it is to restart any entry to correct errors.

After the complete name and address has been entered, the program should ask the operator if he or she wishes to make any changes; then the program should ask which field needs to be changed. Subroutines to ask both of these questions were given earlier in this chapter. We are going to use modified versions of these subroutines, where the calling program provides the question to be asked of the operator. Here is the complete program with added statements beginning at line 250:

```
10 REM NAME AND ADDRESS DATA ENTRY
20 REM DISPLAY THE DATA ENTRY FORM
30 PRINT"■■ ENTER NAME AND ADDRESS"
40 PRINT" ■1■   NAME:"
50 PRINT" ■2■ STREET:"
60 PRINT" ■3■   CITY:"
70 PRINT" ■4■  STATE:";TAB(28);"■5■ ZIP:"
80 REM GET 20 CHARACTER NAME
90 LN%=20
100 PRINT"■■■";TAB(10);
110 GOSUB 8000:NA$=CC$
120 REM GET 20 CHARACTER STREET
130 PRINTCHR$(13);TAB(10);
140 GOSUB 8000:SR$=CC$
150 REM GET 20 CHARACTER CITY
160 PRINTCHR$(13);TAB(10);
170 GOSUB 8000:CI$=CC$
180 REM GET 18 CHARACTER STATE
185 LN%=18
190 PRINTCHR$(13);TAB(10);
200 GOSUB 8000:ST$=CC$
210 REM GET 5 CHARACTER ZIP CODE
220 LN%=5
230 PRINTTAB(34);
240 GOSUB 8000:ZI$=CC$
250 REM ASK IF ANY CHANGES ARE TO BE MADE
260 QU$="DO YOU WANT TO MAKE ANY CHANGES? "
270 PRINT"■■■■■■■■";
280 GOSUB 3000
290 IF YN$="N" THEN STOP
300 REM ASK WHICH FIELD IS TO BE CHANGED
310 QU$="ENTER CHANGE FIELD NUMBER (1 TO 5): "
320 LO%=1:HI%=5
330 GOSUB 3500
340 ON NM% GOTO 400,450,500,550,600
400 REM CHANGE NAME
410 PRINT"■■■";TAB(10);:LN%=20
420 GOSUB 8000:NA$=CC$
430 GOTO 260
450 REM CHANGE STREET
460 PRINT"■■■■";TAB(10);:LN%=20
470 GOSUB 8000:SR$=CC$
480 GOTO 260
500 REM CHANGE CITY
510 PRINT"■■■■■";TAB(10);:LN%=20
520 GOSUB 8000:CI$=CC$
530 GOTO 260
550 REM CHANGE STATE
560 PRINT"■■■■■■";TAB(10);:LN%=18
570 GOSUB 8000:ST$=CC$
580 GOTO 260
600 REM CHANGE ZIP
610 PRINT"■■■■■■";TAB(34);:LN%=5
620 GOSUB 8000:ZI$=CC$
630 GOTO 260
2000 REM CLEAR THE ROW ON WHICH THE CURSOR IS CURRENTLY POSITIONED
2010 PRINTCHR$(13);"■";:REM MOVE CURSOR TO COLUMN 0
2020 FOR I=1 TO 39:PRINT" ";:NEXT
2030 PRINTCHR$(13);"■";
2040 RETURN
3000 REM ASK A QUESTION AND RETURN A RESPONSE OF Y OR N IN YN$
3010 GOSUB 2000
3020 PRINTQU$;
3030 GET YN$:IF YN$<>"N" AND YN$<>"Y" THEN 3030
3040 PRINTYN$;
```

```
3050 RETURN
3500 REM ASK FOR A NUMERIC SELECTION
3510 REM RETURN SELECTION IN NM%
3520 REM NM% MUST BE LESS THAN HI% AND MORE THAN LO%
3530 REM CALLING PROGRAM MUST SET HI%,LO% AND QU$,THE QUESTION ASKED
3540 GOSUB 2000
3550 PRINTQU$;
3560 GET NM$:IF NM$="" THEN 3560
3570 NM%=VAL(NM$)
3580 IF NM%<LO% OR NM%>HI% THEN 3560
3590 PRINTNM$;
3600 RETURN
8000 REM ENTER STRING DATA INTO A FIELD WITH LN% CHARACTERS
8010 REM THE CURSOR MUST BE IN THE FIRST CHARACTER POSITION OF THE FIELD
8020 REM THE RETURN KEY WILL END DATA ENTRY INTO THE FIELD
8030 REM THE ← KEY WILL RESTART DATA ENTRY INTO THE FIELD
8040 REM NO VALIDITY CHECKS ARE MADE ON ANY ENTERED DATA
8050 REM THE ENTERED STRING IS RETURNED IN STRING VARIABLE CC$
8060 ST%=POS(X):REM GET FIELD FIRST CHARACTER POSITION
8070 PRINT"■";:REM REVERSE ENTRY FIELD
8080 FOR I=1 TO LN%:PRINT" ";:NEXT
8090 PRINT"■";CHR$(13);"□";TAB(ST%);
8100 REM ENTER DATA AND DISPLAY AS ENTERED
8110 CC$="":J%=0
8120 FOR I=1 TO LN%
8125 J%=J%+1
8130 GET C$:IF C$="" THEN 8130
8140 IF C$="←" THEN PRINTCHR$(13);"□";TAB(ST%);:GOTO 8070
8150 IF C$=CHR$(13) THEN 8200
8160 PRINTC$;:CC$=CC$+C$
8170 NEXT
8190 REM FILL THE REST OF CC$ WITH BLANKS AND DISPLAY IT
8200 IF J%=LN% THEN 8300
8210 FOR I=J% TO LN%
8220 CC$=CC$+" "
8230 NEXT
8300 PRINTCHR$(13);"□";TAB(ST%);CC$;
8310 RETURN
```

Enter the entire name and address program and run it. If it does not work, check for program errors. Here are some tips when looking for errors:

1. If the display scrolls off the top of the screen, you forgot to terminate the PRINT statement with a semicolon in the subroutine that clears a line.
2. If a reverse field is displayed in the wrong place, you have the wrong number of CURSOR DOWN shifts in a PRINT statement, or you have tabbed to the wrong column, or you have forgotten to separate two items in a PRINT statement with a semicolon.
3. If no message appears at the bottom of the display, make sure that the label you used in the main program to create the display is exactly the same as the label referenced in the subroutine which asks a question.

You should study the name and address program carefully and understand the data entry aids which have been included. They are:

1. By reversing the field into which data must be entered, you clearly indicate to the operator what data is expected, and how many characters are available.
2. When an operator enters a change field number, the reverse field display again quickly tells the operator whether the correct selection was made.
3. An operator does not have to fill in all the characters of a field; when the operator presses the RETURN key the balance of the field is filled with blank characters.
4. At any time the operator can restart entry into a field by pressing the CURSOR LEFT key.

5. When questions are asked, only meaningful character responses are recognized: Y or N for "yes" and "no," or a number between 1 and 5 to select a field. It is very bad programming practice to recognize any key other than a meaningful one. For example to recognize Y for "yes" and any other character for "no" could be disastrous, since accidentally tapping a key could take the operator out of the current data entry prematurely. Conversely, recognizing N for "no" and any other character for "yes" would cause the operator to unnecessarily reenter data into some field, just because the operator accidentally touched the wrong key.

Here are some data entry precautions which we have not taken but could add:

1. Check the ZIP code for any non-digit entry. Similar codes outside the USA do allow alphanumeric entries, however.
2. Many cautious programmers will ask the question ARE YOU SURE? when an operator types "no" in response to the question DO YOU WANT TO MAKE ANY CHANGES? This gives the operator a second chance in the event that he or she accidentally touched the wrong key.
3. We might add an additional key which aborts a current data entry and restores the prior value. For example, if the operator presses the wrong number to select a field which must be changed, the current program forces the operator to re-enter the field. We could easily add another key which aborts the current data entry and retains the previous entry.

Try modifying the name and address entry program yourself to add the additional safety features described above. Also, if you have a CBM 8000 computer, try using its TAB SET capabilities instead of the TAB functions.

PROGRAMMING DISPLAYS AND PRINTOUTS

When you power up a CBM computer, output is directly to the display. You must execute appropriate statements to send the output to the printer or any other device capable of receiving output.

There are a number of differences in the programming techniques required to create a screen display as compared to a hard copy printout. For example, the printer may be wider than the display, in which case output which will fit on a printed line will run over the display line. But there are also significant differences in programming logic which you must use to format a printout as compared to a screen display. This is because cursor control keys can be used to move the cursor around the screen display, but they cannot be used to move a print head around a piece of paper.

There are also many similarities in the programming techniques used to create printouts and displays. **The discussion that follows applies to displays only. If you are planning to write programs that generate output at a printer you should read the discussion of display outputs given in this chapter, and then proceed to the discussion of printer programming given in Chapter 6.**

Programming display output is much simpler than programming data input, since there is no operator interaction to worry about. You must make sure that the display is easy to read, and that is all. Here are a few rules to follow:

1. Avoid crowding too much information into a very small space.

2. If numbers or character strings are listed in columns, align the data so that the eye can quickly run down the column.
3. Use reverse fields on displays to highlight key information, top heading, and/or side headings. Do not reverse fields on printouts; the printer generates very illegible reverse fields.

Below are some common mistakes which you should be aware of, and therefore avoid, when programming displays:

1. Remember to follow individual items in a PRINT statement with a semicolon(;) unless you specifically want the spacing provided by commas (,). This is the most common source of errors in output programming.
2. You will save a lot of programming time if you first get a piece of graph paper, section off rows and columns, then draw the display before attempting to program it. This will allow you to compute rows and columns accurately. The alternative is to use trial and error, which in the end will take a lot more time than drawing the display first.
3. Watch for array subscripts which do not divide evenly into columns. For example, suppose you have 25 items in array N$(I) which you are printing in 3 columns. You might be tempted to generate the display as follows:

```
100 FOR I=1 TO 25 STEP 3
200 REM PROCESS COLUMN 1
 .
 .
 .
300 REM PROCESS COLUMN 2
 .
 .
 .
400 REM PROCESS COLUMN 3
 .
 .
 .
500 NEXT I
```

But on the final pass of the FOR-NEXT loop, indexes 26 and 27 will be computed, although they do not exist. You can easily check for the end of an array in a FOR-NEXT loop as follows:

```
100 FOR I=LO TO HI STEP ST
 .
 .
 .
350 I=I+1
360 IF I>HI THEN 500
 .
 .
 .
500 NEXT
```

An important warning applies to data which you read from a disk file (using techniques which we will describe in Chapter 6). **CBM computers have a nasty habit of adding blank characters onto the end of string variables which are read from a disk file.** For example, if you write names to a disk file, knowing in advance that no name has more than 20 characters, you might assume that when you read these names back from the disk file, each name will still have 20 characters or less. That is not necessarily the case. Some variable number of additional blank characters may get tacked onto the end of the string variable. This can distort your display or printout by extending a field

beyond the column to which you will next tab. You can avoid this problem by using the LEFT$ function. Therefore a PRINT statement such as:

```
100 PRINT TAB(5);N$(I);TAB(30);N$(I+1)
```

would have to be rewritten as follows:

```
100 PRINT TAB(5);LEFT$(N$(I),29);TAB(30);LEFT$(N$(I+1),20)
```

If a list of variables has unknown string lengths, and you want to convert all variables to some fixed length, then you must add blank characters to the end of short strings, and truncate long strings. This is easily done by the following subroutine:

```
10 REM STRING VARIABLE N$ IS TO BE 20 CHARACTERS LONG
20 REM IF LESS THAN 20 CHARACTERS, ADD TRAILING BLANKS
30 REM IF MORE THAN 20 CHARACTERS, TRUNCATE EXCESS CHARACTERS
40 L%=LEN(N$):REM L%=NUMBER OF CHARACTERS IN N%
50 B$="                    ":REM B$ IS A DUMMY 20 BLANK CHARACTER VARIABLE
60 IF L%>20 THEN N$=LEFT$(N$,20):RETURN:REM N$ IS TRUNCATED
70 IF L%=20 THEN RETURN:REM N$ HAS CORRECT LENGTH
80 N$=N$+LEFT$(B$,20-L%):REM N$ IS SHORT, ADD BLANKS
90 RETURN
```

When dealing with large quantities of data, a very common technique is to create a "window" in which to enter the data. In order to provide a simple demonstration, we will create a double-dimensioned 14 × 50-integer array variable. Each integer in the array will contain a four-digit number which identifies the array coordinates as follows:

```
X%(I,J) = 0I0J
```

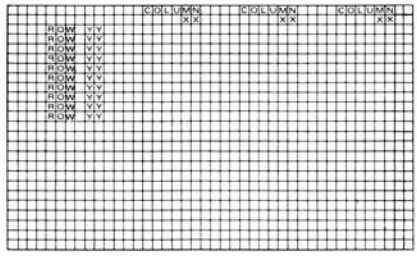
For example:

```
X%(3,2) = 0302
X%(19,8) = 1908
X%(11,12) = 1112
etc.
```

We can create this integer array very simply, as follows:

```
10 DIM X%(14,50)
20 FOR I=1 TO 14
30 FOR J=1 TO 50
40 X%(I,J)=I*100+J
50 NEXT
60 NEXT
```

Now we will display some portion of this array. We will use the top two rows and columns 1 through 10 to create header displays as follows:



XX represents a number in the range 1 through 14
YY represents a number in the range 1 through 50

Here are the necessary program statements to create reverse field row and column headers as illustrated above:

```
1000 REM CREATE ROW AND COLUMN HEADERS
1010 PRINTTAB(9);
1020 FOR I=1 TO 3
1030 PRINT"■    COLUMN■";
1040 NEXT
1050 PRINT CHR$(13);TAB(9);
1060 FOR I=C% TO C%+2
1070 S%=7:IF I<10 THEN S%=8
1080 PRINTSPC(S%);"■";STR$(I);"■";
1090 NEXT
1095 PRINTCHR$(13);
1110 FOR I=R% TO R%+9
1120 S%=1:IF I<10 THEN S%=2
1130 PRINTTAB(2);"■ROW";SPC(S%);STR$(I);"■"
1140 NEXT
1150 RETURN
```

We deliberately create a window that is smaller than the entire screen so that we can better illustrate the concept of a window on data. There is nothing to stop you creating a window that occupies your entire display, however there will be occasions when you want a small window so that concurrent data can appear on the screen.

The STR$ function creates a display that is one character longer than the integer number. This extra character represents the sign. We could remove the sign, but we choose instead to display this extra character in reverse field. But we must account for its presence when counting character positions in order to set the tab on line 1130.

We will now add instructions that ask the operator to enter two numbers representing the smallest column and row of the array. The array element with this column and row number will appear in the top left-hand display position. The display will be filled with adjacent column and row elements, up to the end of the display. Add these lines to your program:

```
5 REM WINDOW ON A TABLE DISPLAY PROGRAM
10 DIM X%(14,50)
20 FOR I=1 TO 14
30 FOR J=1 TO 50
40 X%(I,J)=I*100+J
50 NEXT
60 NEXT
64 PRINT"□";
65 PRINT"■■■■■■■■■■■■■■■■";
70 INPUT "ENTER COLUMN (1 TO 12):";C%
80 IF C%<1 OR C%>12 THEN PRINT"□";:GOTO 70
90 INPUT "ENTER ROW (1 TO 41):";R%
100 IF R%<1 OR R%>41 THEN PRINT"□";:GOTO 90
105 PRINT"□";:GOSUB 1000
110 PRINT"■■■";
120 FOR I=R% TO R%+9
130 PRINT TAB(9);
140 FOR J=C% TO C%+2
150 X$=STR$(X%(J,I))
155 PRINTSPC(10-LEN(X$));X$;
160 NEXT
165 PRINTCHR$(13);
170 NEXT
180 PRINT"■■■CONTINUE? ENTER Y OR N ";
190 GET C$: IF C$<>"Y" AND C$<>"N" THEN 190
200 IF C$="Y" THEN 65
210 STOP
```

Run the program. If you entered it correctly, the first thing you will notice is that the computer stops and appears to do nothing for a while; it is executing the nested FOR-NEXT statements occurring on lines 20 through 60. It takes 10 or 15 seconds to fill array X% with numbers.

The PRINT statement on line 64 clears the screen so that any prior garbage is eliminated before INPUT statements on lines 70 and 90 ask you to enter the beginning row and column numbers. We do not put this clear command into the PRINT statement on line 65, since the program returns to line 65 in order to ask for new column and row numbers, at which time we do not want to erase the prior display.

Note that column numbers from 1 through 12 are allowed; there are three columns, therefore any column number up to 12 will stay within the limit of 14 columns. Row numbers from 1 to 41 are allowed, likewise, since ten columns are displayed, which means that the highest ten column numbers would be 41 through 50.

The integer value from array X% is converted into an ASCII string on line 150 before being printed on line 155. This conversion has been made to simplify display formatting. It is easy to compute the number of spaces between columns, as shown by the PRINT statement on line 155. It is not so easy to align numbers correctly when displaying integers. To prove this for yourself, remove line 150 and change line 155 as follows:

```
155 PRINT SPC(5);X%(J,I);
```

Numbers will align providing you do not display any four-digit numbers, in which case the display will overflow a 40-character screen. If you display three-digit numbers the rows are all shifted over one column to the right. You could correct this discrepancy by increasing the tab on line 130 from 9 to 10. Try it. When you next run the program it will overflow the 40-column display line and give you a lot of extra carriage returns.

Notice that the statements which ask for input on lines 70, 90, and 180 are all followed by program steps that disallow all invalid inputs. Even in this simple demonstration program we take the time to program safe input.

A useful refinement to a program that displays a window on an array is to provide the operator with a means of moving the window up or down, left or right. This is easily done. Using available symbols on a CBM standard keyboard, we will use the spade sign (♠) to move up one row, which means that the beginning row number is decreased by 1. We will use the heart sign (♥) to move down one row, which means that the beginning row number is increased by 1. We will use the less than sign (<) to move the table one column to the left (decrease the beginning column number by 1), and use the greater than sign (>) to move the table one column to the right (increase the beginning column number by 1). To accomplish this task we must replace statements on lines 180 through 210 with the following statements:

```
180 PRINT"■■■CONTINUE? ENTER ♠,♥,<,>,Y OR N ";
190 GET C$: IF C$="" THEN 190
200 REM IF C$=ATN THEN DECREASE ROW BY 1
210 IF C$="♠" THEN R%=R%-1:PRINTCHR$(13);"■";:GOTO 100
220 REM IF C$=COPY THEN INCREASE ROW BY 1
230 IF C$="♥" THEN R%=R%+1:PRINTCHR$(13);"■";:GOTO 100
240 REM IF C$=< THEN DECREASE COLUMN BY 1
250 IF C$="<" THEN C%=C%-1:GOTO 300
260 REM IF C$=> THEN INCREASE COLUMN BY 1
270 IF C$=">" THEN C%=C%+1:GOTO 300
280 REM IF C$=Y,ENTER NEW ROW AND COLUMN IF C$=N,STOP
290 IF C$="Y" THEN 65
295 IF C$="N" THEN STOP
296 GOTO 190:REM REJECT ANY OTHER C$ INPUT
300 IF C%<1 OR C%>12 THEN PRINTCHR$(13);:GOTO 70
310 GOTO 105
```