To print a trailing sign, change the PRINT# statement on line 100 as follows:

```
100 PRINT#2,"9999999.99-"
```

Now run the program. A minus sign appears after negative numbers; positive numbers have no sign printed.

Notice that all numbers are truncated after the specified digit has been printed. The printer does not round up.

Now we will convert numbers to $ amounts by adding a $ sign to the front of the numeric specification. We will also print a leading sign; the PRINT# statement on line 100 must now change as follows:

```
100 PRINT#2,"S$9999999.99"
```

When you re-run the program you will get the following printout:

```
+$      1.75
-$  12300.00
+$       .74
+$     12.00
-$    456.83
+$  23456.78
-$    100.79
+$4789326.00
```

Note that S must precede the $ sign. If a $ precedes the S, unformatted numbers will be printed.

It is common in financial reports to identify negative $ amounts with a trailing minus sign. You can generate such a printout by removing the S and replacing it with a trailing minus sign. Change line 100 as follows:

```
100 PRINT#2,"$9999999.99-"
```

Now re-run the program; you will get the following printout:

```
$      1.75
$  12300.00-
$       .74
$     12.00
$    456.83-
$  23456.78
$    100.79-
$4789326.00
```

In any printout of $ amounts, **the $ sign can be printed directly in front of the first numeric digit; this requires all character positions preceding the decimal point to be filled with $ signs**. Change 100 as follows:

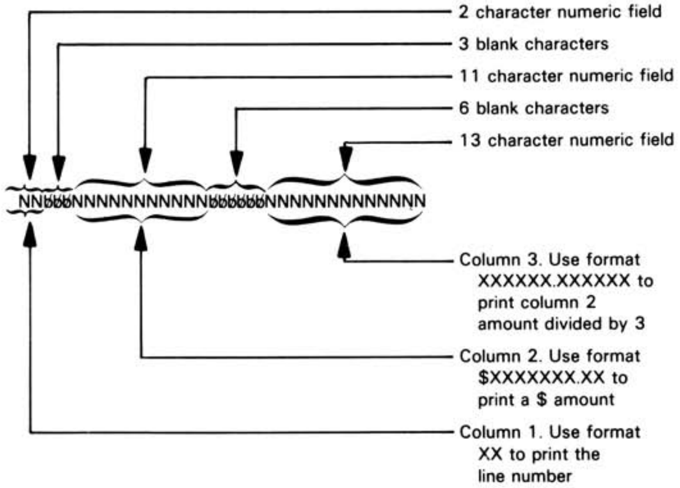
```
100 PRINT#2,"$$$$$$$.99-"
```

Now re-run the program; you will get the following printout:

```
     $1.75
 $12300.00-
      $.74
    $12.00
   $456.83-
 $23456.78
   $100.79-
******.****
```

What went wrong? The eighth number was printed as asterisks. The problem is that the new line 100 has seven $ characters preceding the decimal point; it needs 8. **You need one $ for each character position preceding the decimal point, plus an additional $ to select the $ character printout.**

So far we have printed formatted numeric data in a single column. To print multi-column data, provide a separate numeric format specification for each column using blank spaces to separate numeric specifications. To illustrate multi-column printing consider the following 3-column output:


2 character numeric field
3 blank characters
11 character numeric field
6 blank characters
13 character numeric field
NNb̸b̸b̸NNNNNNNNNNNb̸b̸b̸b̸b̸b̸NNNNNNNNNNNNN
Column 3. Use format XXXXXX.XXXXXX to print column 2 amount divided by 3
Column 2. Use format $XXXXXXX.XX to print a $ amount
Column 1. Use format XX to print the line number

The PRINT# statement on line 100 must change as follows to specify the 3 column format illustrated above:

```
100 PRINT#2,"99   $$$$$$$$.99-    999999.999999"
```

We will change the PRINT# statement on line 130 to print line number I, N, and N/3. Here is the new line 130:

```
130 PRINT#1,I,N,N/3
```

When you run the program the following printout will be generated:

```
 1         $1.75              .583333
 2     $12300.00-         4100.000000
 3          $.74              .248940
 4        $12.00             4.000000
 5       $456.83-          152.277333
 6      $23456.78          7818.926670
 7       $100.79-           33.599333
 8   $4789326.00        ******.******
```

Each column of numbers has been printed according to the specification provided for that column in the formatting PRINT# statement. **The number of spaces separating printed columns is equal to the number of spaces separating the column formats in the PRINT# statement on line 100.**

Printing Formatted Strings

To print formatted strings you use the letter A to identify each string character position. Use space codes to separate fields, if necessary. The entire format is specified as a single string variable appearing in the parameter list of a PRINT# statement. As described earlier, this PRINT# statement must specify a logical file number which was opened to physical unit 4 with secondary address 2.

String variables which are to be printed using the specified format are output using another PRINT# statement whose logical file number was opened specifying physical unit 4 and secondary address 1. **String variables in the PRINT# statement parameter list must be separated by CHR$(29) characters, which may be generated using the CURSOR RIGHT key within a string. Strings are left-justified within the specified field; trailing character positions (if any) are filled with blanks. Leading space codes are truncated.**

Here are two PRINT# statements that print formatted strings:

```
100 PRINT#X,"AAAAAAAAAA      AAAAAAAAAAAA"
110 PRINT#Y,M$CHR$(29)N$
```

X represents any valid logical file number that has been opened specifying physical unit 4 with secondary address 2. Y represents any logical file number that has been opened specifying physical unit 4 with secondary address 1.

The PRINT#X statement specifies 10-character and 12-character string fields separated by five blank spaces.

The PRINT#Y statement specifies two string variables, M$ and N$, separated by the required separator CHR$(29). **Notice that commas have not been used to separate elements of the PRINT#Y statement parameter list.** You can use commas if you wish; the following alternate PRINT#Y statement is valid:

```
PRINT#Y,M$,CHR$(29),N$
```

You can replace M$ and N$ with actual string elements, with or without commas separating the string elements from the CHR$(29) separators. This may be illustrated as follows:

```
PRINT#Y,"ONE"CHR$(29)"TWO"
```

To illustrate formatted string printing, we will modify program NUM.FORM.PRINT to generate STR.FORM.PRINT. The program and sample run are listed below.

```
10 REM PROGRAM "STR.FORM.PRINT"
20 REM DEMONSTRATE FORMATTED STRING PRINTOUT
30 DATA  "MARY PERKINS","35 WEST ST.","BERKELEY","CALIFORNIA","94705"
35 DATA "345-67-8910","SPONSOR","AXC"
70 OPEN 1,4,1:REM OUTPUT DATA VIA LOGICAL FILE 1
80 OPEN 2,4,2:REM OUTPUT DATA FORMATS VIA LOGICAL FILE 2
90 REM OUTPUT DATA FORMAT
100 PRINT#2,"AAAAAAAAAA      AAAAAAAAAAAA"
105 SP$=CHR$(29)
110 FOR I=1 TO 4
120 READ M$,N$
130 PRINT#1,M$,SP$,N$
140 NEXT I
150 CLOSE 1
155 CLOSE 2
160 STOP
```

```
MARY PERKI      35 WEST ST.
BERKELEY        CALIFORNIA
94705           345-67-8910
SPONSOR         AXC
```

The PRINT#X statement appears on line 100 specifying logical file 2, which is opened on line 80. The PRINT#Y statement appears on line 130 specifying logical file 1, which is opened on line 70. Instead of using CHR$(29) in the PRINT#1 statement on line 130, we use SP$, which is equated to CHR$(29) on line 105.

The eight numeric data items which appeared in a single DATA statement in the NUM.FORM.PRINT program now occupy two DATA statements on lines 30 and 35. Eight string variables are specifed; they consist of an arbitrary address followed by a social security number and two code words, shown on line 35 as "SPONSOR" and "AXC".

Note that the first field (containing the name MARY PERKINS) has been truncated after the I of PERKINS. You must add three more A's to the first field specification in order to accommodate the entire name. Notice also that all fields are left justified. **In order to insert leading space codes you cannot use a normal space bar character; you must use CHR$(160), the upper case space bar character.** We can demonstrate this by adding leading blank characters to one string variable; we will choose AXC. Change the data statement on line 35 as follows:

```
35 DATA "345-67-8910","SPONSOR","  AXC"
```

Press space bar twice

Now rerun the program. The printout does not change. The two blank characters preceding AXC were ignored. Now retype the modified data statement, holding the shift key down while you enter the two spaces in front of AXC. This time when you run the program AXC will be shifted two character positions to the right in the printout.

Using string concatenation you can shift string variables to the right within a string field. This is illustrated by the modification of program STR.FORM.PRINT shown below, followed by a sample run.

```
10 REM PROGRAM "STR.FORM.PRINT"
20 REM DEMONSTRATE FORMATTED STRING PRINTOUT
30 DATA  "MARY PERKINS","35 WEST ST.","BERKELEY","CALIFORNIA","94705"
35 DATA "345-67-8910","SPONSOR","AXC"
70 OPEN 1,4,1:REM OUTPUT DATA VIA LOGICAL FILE 1
80 OPEN 2,4,2:REM OUTPUT DATA FORMATS VIA LOGICAL FILE 2
90 REM OUTPUT DATA FORMAT
100 PRINT#2,"AAAAAAAAAA     AAAAAAAAAAAA"
105 SP$=CHR$(29)
106 BL$="            ":REM 12 UPPER CASE SPACE CODES
110 FOR I=1 TO 4
120 READ M$,N$
125 IF LEN(M$)<10 THEN M$=LEFT$(BL$,(10-LEN(M$)))+M$
126 IF LEN(N$)<12 THEN N$=LEFT$(BL$,(12-LEN(N$)))+N$
130 PRINT#1,M$SP$N$
140 NEXT I
150 CLOSE 1
155 CLOSE 2
160 STOP
```

```
MARY PERKI    35 WEST ST.
  BERKELEY     CALIFORNIA
     94705    345-67-8910
   SPONSOR            AXC
```

In order to right-justify string fields, statements on lines 125 and 126 check for string variables that are shorter than the specified field width. Lengths for shorter variables are increased to the field width by adding leading upper-case space characters. Leading upper-case space characters are taken from string variable BL$, which is defined on line 106. The number of upper-case space characters is computed as the difference between the field width and the length of the string variable. This number of characters is taken from BL$ using the LEFT$ function.

We will now modify program STR.FORM.PRINT to print data using a reasonable format. For example, the five name and address fields might be printed in a single vertical column (with no truncated characters), while the three additional fields are printed on a single line below the name and address. Program STR.FORM.PRINT1, listed below, generates the required printout. A sample run is shown after the listing.

```
10 REM PROGRAM "STR.FORM.PRINT1"
20 REM DEMONSTRATE FORMATTED STRING PRINTOUT
30 DATA  "MARY PERKINS","35 WEST ST.","BERKELEY","CALIFORNIA","94705"
35 DATA "345-67-8910","SPONSOR","AXC"
70 OPEN 1,4,1:REM OUTPUT DATA VIA LOGICAL FILE 1
80 OPEN 2,4,2:REM OUTPUT DATA FORMATS VIA LOGICAL FILE 2
90 REM OUTPUT DATA FORMAT
105 SP$=CHR$(29)
110 FOR I=1 TO 8
120 READ M$(I)
140 NEXT I
150 PRINT#2,"AAAAAAAAAAAAAA"
160 FOR I=1 TO 5
170 PRINT#1,M$(I)
180 NEXT I
190 PRINT#2,"AAAAAAAAAAA   AAAAAAA   AAA"
200 PRINT#1,M$(6)SP$M$(7)SP$M$(8)
210 CLOSE 1
220 CLOSE 2
230 STOP
```

```
MARY PERKINS
35 WEST ST.
BERKELEY
CALIFORNIA
94705
345-67-8910   SPONSOR   AXC
```

All eight string variables have been read into the string array M$(I) by the FOR-NEXT loop on lines 110 through 140, before any string data is printed out. Five fields are then printed in a single vertical column by the FOR-NEXT loop on lines 160 through 180, using the format specified by the PRINT# statement on line 150. A new format is then specified by the PRINT# statement on line 190; this new format is used to print out the last three string variables using the PRINT# statement on line 200.

Printing Mixed Formatted Data

You can mix numeric and string data in formatted printer output. A simple demonstration of such output is given by program STR.FORM.PRINT2, which is listed below together with a sample printout.

```
10 REM PROGRAM "STR.FORM.PRINT2"
20 REM DEMONSTRATE FORMATTED STRING PRINTOUT
30 DATA  "MARY PERKINS","35 WEST ST.","BERKELEY","CALIFORNIA","94705"
35 DATA "345-67-8910","SPONSOR","AXC"
70 OPEN 1,4,1:REM OUTPUT DATA VIA LOGICAL FILE 1
80 OPEN 2,4,2:REM OUTPUT DATA FORMATS VIA LOGICAL FILE 2
90 REM OUTPUT DATA FORMAT
105 SP$=CHR$(29)
110 FOR I=1 TO 8
120 READ M$(I)
140 NEXT I
150 PRINT#2,"99  AAAAAAAAAAAAAA"
160 FOR I=1 TO 5
170 PRINT#1,I,M$(I)
180 NEXT I
190 PRINT#2,"99  AAAAAAAAAAA   AAAAAAA   AAA"
200 PRINT#1,I,M$(6)SP$M$(7)SP$M$(8)
```

```
210 CLOSE 1
220 CLOSE 2
230 STOP

1  MARY PERKINS
2  35 WEST ST.
3  BERKELEY
4  CALIFORNIA
5  94705
6  345-67-8910    SPONSOR   AXC
```

This program is a minor variation of STR.FORM.PRINT1. A line number numeric followed by three blank spaces has been added to the two PRINT# statements on lines 150 and 190. The data output PRINT# statements on lines 170 and 200 each print the FOR-NEXT loop index.

A second program, PRINTDATE, is more interesting. It accepts the month, day and year entered at the keyboard as three separate numeric variables. Each date is printed with a dash separating month from day and day from year. Program PRINTDATE is listed below together with a sample printout for three dates.

```
10 REM PROGRAM "PRINTDATE"
20 OPEN 1,4,1:REM OUTPUT DATA VIA LOGICAL FILE 1
30 OPEN 2,4,2:REM OUTPUT DATA FORMAT VIA LOGICAL FILE 2
40 PRINT"♥QQ"
50 INPUT "ENTER MONTH:";M
60 INPUT "ENTER DAY  :";D
70 INPUT "ENTER YEAR :";Y
80 PRINT#2,"AAAAA   99A99A99"
90 SP$=CHR$(29)
100 PRINT#1,"DATE:"SP$,M,"-"SP$,D,"-"SP$,Y
110 PRINT"ANOTHER DATE? ENTER Y FOR YES OR N FOR NO";
120 GET YN$:IF YN$="" THEN 120
130 IF YN$="N" THEN PRINTYN$:STOP
140 IF YN$<>"Y" THEN 120
150 GOTO 40
```

```
DATE:    6-12-80
DATE:   12-25-81
DATE:    1- 1-70
```

Program PRINTDATE makes no validity checks on the numbers entered for month, day and year since we want to focus attention on printer formatting rather than good data entry programming practice. But the usefulness of formatted printout is obvious from the example below.

Including Literals in Formatted Printout

The printer format specification can include literal characters. A literal character is printed exactly as it appears in the printer format specification; it does not specify format for data occurring in a subsequent PRINT# statement. **A literal character must be preceded by the REVERSE ON (RVS) character.** The character coming directly after the REVERSE ON is printed normally. In consequence you cannot print reverse field literal characters.

Program PRINTDATEL1 makes very simple use of literals. A literal dash separates month from day and day from year, replacing the string used by program PRINTDATE. To create program PRINTDATEL1, load program PRINTDATE from the previous section, then change the PRINT# statements on lines 80 and 100 as shown below. PRINTDATEL1 and PRINTDATE generate the same display and printout.

```
10 REM PROGRAM "PRINTDATEL1"
20 OPEN 1,4,1:REM OUTPUT DATA VIA LOGICAL FILE 1
30 OPEN 2,4,2:REM OUTPUT DATA FORMAT VIA LOGICAL FILE 2
40 PRINT"]QQ"
50 INPUT "ENTER MONTH:";M
60 INPUT "ENTER DAY   :";D
70 INPUT "ENTER YEAR :";Y
80 PRINT#2,"AAAAA   99]-99]-99"
90 SP$=CHR$(29)
100 PRINT#1,"DATE:"SP$,M,D,Y
110 PRINT"ANOTHER DATE? ENTER Y FOR YES OR N FOR NO";
120 GET YN$:IF YN$="" THEN 120
130 IF YN$="N" THEN PRINTYN$:STOP
140 IF YN$<>"Y" THEN 120
150 GOTO 40
```

You can create forms, while printing output, by making appropriate use of literals in printer format statements. However, literals and text must come from the same character set. Moreover, the printers recognize the PET character sets. When using CBM computers, therefore, it is very difficult to generate forms using literals. But a program written on a 2001 computer can be run on a CBM computer in order to generate forms.

SPECIAL PRINTER CONTROL CHARACTERS

There are a number of special printer control characters which modify printer output when inserted in data. Printer control characters are summarized in Table 6-4.

Printer control characters are inserted in the data stream transmitted to the printer via secondary address 0 or 1. Printer control characters are not transmitted as part of the format specified using secondary address 2.

You can use printer control characters with formatted or unformatted printouts.

The first two entries in Table 6-4, CHR$(29) and CHR$(160), must be used with formatted printouts (as previously described); they are ignored in unformatted printouts.

Codes listed as optional in Table 6-4 can be used with formatted or unformatted printouts; their effect is the same in either case.

Enhanced Character Printout

CBM printers normally generate characters using a dot matrix that is seven dots high and six dots wide. **If you include a CHR$(1) character** within a data output PRINT# statements parameter list, **all characters following the CHR$(1) are printed double-width:** using a dot matrix that is seven dots high and 12 dots wide. More than one CHR$(1) character can appear in a single parameter list. Each CHR$(1) character takes the previous character width and doubles it. Following two CHR$(1) characters, therefore, 7 by 24 dot matrices will be used to print characters. After a third CHR$(1) character, 7 by 48 dot matrices would be used.

In order to demonstrate enchanced printout, load program STR.FORM.PRINT1 and add the following line:

```
125 M$(I)=CHR$(1)+M$(I)
```

When you run this modified program, the first printed column (including name, address and social security number) is printed using double-width characters. The word SPONSOR uses quadruple-width characters, while the letters AXC are printed using characters that are eight times normal width. Here is a sample printout:

```
MARY PERKINS
35 WEST ST.
BERKELEY
CALIFORNIA
94705
345-67-8910      SPONSOR      AXC
```

What happened?

Line 125 added an enchancement character to the beginning of each string variable. Therefore the first string variable on any line is printed double-width, the second string variable is printed quadruple-width and the third variable is printed using characters that are eight times standard width.

You do not have to concatenate CHR$(1) characters to strings. **You can insert CHR$(1) into the PRINT# statement parameter list, but you must not use commas to separate CHR$(1)**. For example, reload program STR.FORM.PRINT1, and replace line 200 with these two lines:

```
195 E$=CHR$(1)
200 PRINT#1,E$M$(6)SP$E$M$(7)SP$E$M$(8)
```

When you run this program, the name and address are printed using standard character widths. The social security number is printed using double-character widths, the word SPONSOR is printed using quadruple-character width, while AXC is printed using characters that are eight times normal width. Here is a sample printout:

```
MARY PERKINS
35 WEST ST.
BERKELEY
CALIFORNIA
94705
345-67-8910      SPONSOR      AXC
```

You can print enhanced numeric variables. The numeric variable is included in the PRINT# statement parameter list, but it must have commas separating it from other variables. To demonstrate enhanced numeric printout we will again start with program STR.FORM.PRINT1. Modify lines 190 through 200 as follows:

```
190 PRINT#2,"AAAAAAAAAAAA   99999  AAAAAAA"
195 E$=CHR$(1)
196 N=12345
200 PRINT#1,E$M$(6)SP$,N,E$M$(7)
```

The final line printer format has been changed by the PRINT# statement on line 190; two string fields are printed with a numeric field appering between them. The PRINT# statement on line 200 specifies M$(6) and M$(7) as the two string fields, with the new numeric variable N between them. N is equated to 12345 on line 196. In the parameter list of the PRINT# statement on line 200 **notice that the numeric variable N is separated using commas, but commas are not used to separate string variables**.

These syntax rules are very specific and must be observed in order to generate successful mixed, enhanced numeric and string printout. Here is a sample of the printout generated by STR.FORM.PRINT1 with lines 190 through 200 modified as listed above:

```
MARY PERKINS
35 WEST ST.
BERKELEY
CALIFORNIA
94705
345-67-8910     12345    SPONSOR
```

You can cancel character enhancement using the CHR$(129) character. Subsequent characters revert to standard size until another CHR$(1) character is encountered.

Printing Reverse Field Characters

Reverse field characters can be included in a PRINT# statement parameter list using the RVS ON and RVS OFF keys. However, you should not print more than five consecutive lines of reverse field characters; if you do, the printhead will wear out very quickly.

Printing Control Characters

To print a quote character you must use CHR$(34).

If you print a single CHR$(34), or any odd number of quote characters in this fashion, then the printer will subsequently display all control characters via their graphic representation.

The only time you are likely to do this is when you are listing programs which include control characters that would not normally be printed.

PAGE FORMAT

Number of Lines per Page

Unless otherwise instructed, CBM printers pay no attention to page length. To enable paging, transmit the CHR$(147) character to the printer as data. The printer then assumes a 66-line page; it prints 60 lines, skips six lines, prints another 60 lines, and so on. Below is the listing for a simple program that turns paging on, then prints a line number followed by the character string ABCDEFG. If you enter and run this program, you will see paged printing in action.

```
10 REM PROGRAM "PAGING" TESTS PAGING OPTIONS
20 OPEN 1,4:REM OPEN UNFORMATTED PRINTOUT
30 REM SELECT PAGING
40 PRINT#1,CHR$(147)
50 FOR I=1 TO 100
60 PRINT#1,I,"ABCDEFG"
70 NEXT I
80 CLOSE 1
90 STOP
```

You can change the number of lines printed per page once paging has been enabled. To do this, you output the selected number of lines as numeric data to a logical file which must be opened specifying physical unit 4 with secondary address 3. The printer then assumes that the page length equals the number of lines specified, plus six. The specified number of lines are printed on each page, with six skipped lines between each page. Program PAGINL25, listed below, prints 25 lines per page.

```
10 REM PROGRAM "PAGINGL25" TESTS PAGING OPTIONS
20 OPEN 1,4:REM OPEN UNFORMATTED PRINTOUT
25 OPEN 3,4,3:REM OPEN FILE TO SELECT NUMBER OF LINES PER PAGE
30 REM SELECT PAGING
40 PRINT#1,CHR$(147)
45 PRINT#3,25:REM SELECT 25 LINES PER PAGE
50 FOR I=1 TO 100
60 PRINT#1,I,"ABCDEFG"
70 NEXT I
80 CLOSE 1
85 CLOSE 3
90 STOP
```

The PRINT# statement on line 45 specifies 25 lines per page. Logical file 3 is opened on line 25.

You can change the number of printed lines per page by outputting a new value to secondary address 3. The new value goes into effect at the beginning of the next page; the current page is printed using the old number of lines per page.

Add the following line to program PAGINGL25:

```
55 IF I=23 THEN PRINT#3,10
```

Run the program twice. The first time a 25-line page is printed, followed by a number of ten-line pages. But on the second execution something strange happens; a ten-line page is printed, followed by a 25-line page, and then a number of ten-line pages. The printer remembered the previously specified number of lines per page and used it for the first page of the new run.

Top of Form

While paging is in effect, if you print a CHR$(19) character, the printer will skip remaining lines on the current page, and position itself at the first print line of the next page. Printing continues from this new position. This is referred to as a top of form. If a page does not print to the last line (and this is the rule rather than the exception), you should end the page by printing a top of form; this will advance the printer to the next page. You do not have to count remaining lines and skip over them.

Space Between Lines (Model 2022)

The model 2022 printer allows you to change the space between printed lines. Printers divide each vertical inch into 144 steps. Normally each line is allotted 24 steps. Thus six lines are printed per vertical inch. **The model 2022 line printer allows you to change the number of lines that will be printed per vertical inch.** To do this, you must open a logical file specifying physical unit 4 with secondary address 6. Then output a CHR$ function to this logical file number, specifying the new number of steps per line as the CHR$ function's argument.

Suppose you want to print eight lines per inch; the number of steps per inch then becomes 144/8, which equals 18. Here are the statements needed to make this change:

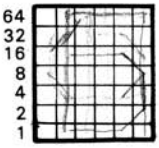
```
10 OPEN 6,4,6
20 PRINT#6,CHR$(18)
```

If you have a model 2022 printer, load program PAGINGL25, insert these two lines, then run the program. Lines will be printed with no space in between them; the vertical width of characters does not change when you increase the number of lines per inch. Steps are removed (or inserted) between lines. By specifying appropriate steps per line you can print lines that overlap, or have a lot of space between them.

DEFINING YOUR OWN CHARACTERS

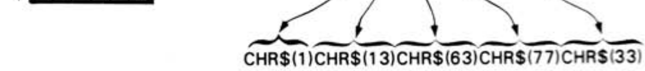
CBM printers allow you to define, or draw, your own printer characters.

All printer characters are generated using a 7 × 6 dot matrix. To create your own character draw 7 × 6 dot matrix as follows:



Each row in the dot matrix is represented by a number, ranging from 1 to 64. The top row has the value of 64, while the bottom row has the value 1. (Each row value is double the previous row value.)

Now generate your character by drawing dots in the 7 × 6 matrix. Here is an English pound character:



You must now convert the character into 6 numbers. Each number corresponds to 1 column of the 7 × 6 matrix and identifies the dots in that column. The first of the 6 numbers represents the left-most column and the last of the 6 numbers represents the right-most column.

To compute the number for any column, write down row values corresponding to each existing dot, then sum the row values, as illustrated above.

Next the six numbers must be converted into a six-character string; each character of the string is a CHR$ function, where the column total becomes the CHR$ function argument. Thus the English pound character becomes a six-character string where the first character has the value CHR$(1), the second character has the value CHR$(13), the third character has the value CHR$(63), the fourth and fifth characters both have the values CHR$(77), and the sixth character has the value CHR$(33). This string is output to the printer using a PRINT# statement that specifies a logical file opened with physical unit 4 and secondary address 5. The printer stores the special character; it does not print it. Subsequently any PRINT# statement that prints data specifies the special character using the function CHR$(254).

The steps needed to print a special character are illustrated by program POUNDCHAR listed below. This program, when executed, will print a column of ten English pound signs.

```
10 REM PROGRAM "POUNDCHAR"
20 REM DEMONSTRATE SPECIAL PRINTER CHARACTER GENERATION
30 DATA 1,13,63,77,77,33
35 EP$=""
40 OPEN 1,4:REM OPEN PRINTER
50 OPEN 5,4,5:REM OPEN SPECIAL CHARACTER GENERATION FILE
60 FOR I=1 TO 6
70 READ EP
80 EP$=EP$+CHR$(EP)
90 NEXT I
95 PRINT#5,EP$
100 FOR I=1 TO 10
110 PRINT#1,CHR$(254)
120 NEXT I
130 CLOSE 1
140 CLOSE 5
150 STOP
```

Let us examine how the pound sign is created and printed.

The data statement on line 30 specifies the number and location of dots in the character matrix, as illustrated previously.

The FOR-NEXT loop on lines 60 through 90 generate the six-character string representing the pound sign and assign this string to string variable EP$. Each number from the data statement is read into numeric variable EP by the READ statement on line 70; this numeric value is converted into a character, and a character is concatenated to EP$ on line 80. The assembled string is output to logical file 5 on line 95. Logical file 5 was opened on line 50 specifying physical unit 4 and secondary address 5. After the PRINT# statement on line 95 has been executed, the printer holds one special character, which it recognizes and prints on encountering a CHR$(254) function in the data string received from a PRINT# statement. This occurs each time the PRINT# statement on line 110 is executed.

Note that the CBM printer can only recognize one special character at any time. You can change the special character by creating a new 6 character string and outputting this string to the printer via secondary address 5. Although this technique is quite straightforward, it does not readily lend itself to the indiscriminate use of the many special characters.

Using Special Characters to Print Non-Dollar Monetary Data

The $ sign is not much use when printing financial data outside of the USA and Canada. Some other character must be substituted for the $ sign. This is easily done using formatted printout in conjunction with special character generation.

Program POUNDVAL, listed below, uses the English pound character which we just generated to print English financial data with a trailing sign. Two sample printouts are shown at the end of the listing.

```
10 REM PROGRAM "POUNDVAL"
20 REM PRINT A NUMERIC VALUE AS BRITISH POUNDS
30 REM CREATE THE POUND SIGN
40 DATA 1,13,63,77,77,33
50 OPEN 5,4,5
60 EP$=""
70 FOR I=1 TO 6
80 READ EP
90 EP$=EP$+CHR$(EP)
100 NEXT I
110 PRINT#5,EP$
```

```
120 OPEN 1,4,1:REM USE FORMATTED PRINTOUT
130 OPEN 2,4,2
140 REM OUTPUT ENGLISH POUND PRINT FORMAT
150 PRINT#2,"AAAAAA   A999999.99-"
160 INPUT "ENTER AMOUNT:";N
170 PRINT#1,"VALUE="CHR$(29)CHR$(254)CHR$(29),N
180 CLOSE 1
190 CLOSE 2
200 CLOSE 5
210 STOP

VALUE=   £  1234.56
VALUE=   £  1234.56-
```

The pound sign is created by statements on lines 40 through 110. These statements have been taken from program POUNDCHAR.

The OPEN statement on lines 120 and 130 open logical files 1 and 2 for formatted printout. The format is output by the PRINT# statement on line 150; a six-character string field if specified, followed by three blank spaces and then a numeric field with preceding single character string field. The numeric field has two places after the decimal point and a trailing sign

The INPUT statement on line 160 lets you enter a number which is assigned to numeric variable N. N is printed by the PRINT# statement on line 170.

Let us examine this PRINT# statement parameter list.

The string "VALUE=" is printed in the first 6 character string fields. This character is followed by the mandatory string separator CHR$(29). Three spaces are printed as required by the printer format. Next comes a single character string field. The character is CHR$(254); it is followed by the mandatory CHR$(29) string field terminator. CHR$(254) selects the special character. The pound sign is therefore printed in front of the numeric field. Numeric variable N is printed in the numeric field.

PRINTER DIAGNOSTIC MESSAGES

If you are having problems with printer output, enable a logical file selecting physical unit 4 with secondary address 4. This will cause the printer to output detailed diagnostic messages when it encounters identifiable errors in printout specifications. You do not have to execute any statements in order to generate error diagnostics; they are output automatically.

Programs in their final form will not normally use printer diagnostic messages. These diagnostic messages are used while you are writing a program, in order to find errors.

You can create a sample diagnostic message by loading program STR.FORM.PRINT1 into memory. Change one of the A format specifications on line 190 to some illegal character such as Q. Then add the following line:

```
85 OPEN 4,4,4
```

When you run the program an error message similar to the one shown below will be generated.

```
MARY PERKINS
35 WEST ST.
BERKELEY
CALIFORNIA
94705
345-6
AAAAAQAAAAA   AAAAAAA   AAA
     ↑
*****BAD FORMAT*****
PONSOR AXC
```