INKEY$

The inkey function.

FORMAT

INKEY$

NOTES

INKEY$ reads the keyboard and returns whatever value it finds there. If you press a key at the moment that INKEY$ reads the keyboard, the function returns that key's value as a one-character string. It does not display that character on the screen. Instead, it passes the string to your program. When INKEY$ reads the keyboard and finds nothing, it returns a null string (length zero).

NOTE: INKEY$ does not of itself wait for a keypress to occur. If you want to monitor the keyboard continuously, you must put INKEY$ in a loop (see example below).

EXAMPLE

```
1000 LET Loop=0
1100 LET Key$=INKEY$
1200 IF Key$="" THEN Loop = Loop+1:PRINT " "; Loop; " "; ELSE PRINT
"  ***  "; Key$; "  ***  ";
1300 GOTO 1100
1400 END
```

This example prints a sequential number on the screen each time INKEY$ reads the keyboard. When you press a key, it prints the key surrounded on either side by " *** ".

INPUT

The INPUT statement asks the user to enter data. It then assigns the data to specifed variables.

FORMAT

```
INPUT [;]["promptString"] {;|,} variablesList
```

NOTES

When an INPUT statement executes, it prints the contents of the promptString. If you follow the promptString with a semicolon, a question mark will follow this string. For example,

```
     1000 INPUT "Your name"; Name$
```

prints on the screen as

```
     Your name?
```

If you put a comma at the end of the string no question mark appears. If you do not include a promptString, the program only displays the question mark. You must enclose the promptString in quotation marks; it can contain any printable characters.

Program execution stops after displaying promptString and question mark (if specified). Execution waits for you to enter data and press CODE-RETURN. If you place a semicolon directly after the word INPUT, the cursor will remain on the same line as the user's response after confirming.

Multiple variables must appear at the end of the INPUT statement. You cannot place variables within the input string. The following example places a variable (Count) in the input string to describe a range of choices. This is an illegal statement.

```
     1500 INPUT "Pick a number (from 1 to ";Count; ") "; Choice
```

The INPUT statement wants to put your data into Count, because Count comes at the end of a prompt string. To put such an informational variable in an INPUT statement, write two lines, one a PRINT statement, the other an INPUT statement. In the example below, we break the illegal line 1500 into two lines. The semicolon at the end of line 1500 causes the two to print like one statement.

```
     1500 PRINT "Pick a number from 1 to ";Count;
     1600 INPUT Choice
```

Data entered via the keyboard is assigned to the variable(s) specified in the variablesList. The number of data items entered must be equal to the number of variables specified in variablesList. You must separate multiple variables in variablesList with commas.

Each data item entered must be of the same type as that specified by the corresponding variable name. The variable names in variablesList can be any mix of numeric and string variable names including subscripted variables. However, each input must be of the same kind as its variable.

NOTE: INPUT does not accept a comma or a semicolon as valid input. You must start your string with the double quotation mark (") if you want to include either of these characters.

If you respond with the wrong kind of constant (giving letters to a numeric variable or including a comma or semicolon in an input string, for example), you will see the message

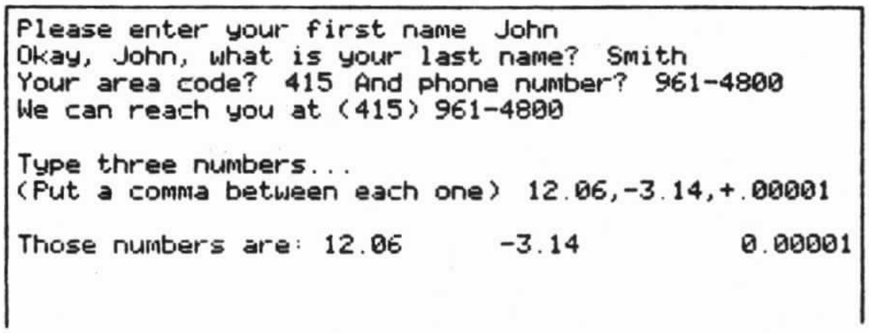
```
Invalid input: Re-enter data
```

EXAMPLE

```
1000 INPUT "Please enter your first name", First$
1100 PRINT "Okay, "; First$;
1200 INPUT ", what is your last name"; Last$
1300 INPUT; "Your area code"; Area$
1400 INPUT " And phone number"; Phone$
1500 PRINT "We can reach you at ("; Area$; ")"; " ";Phone$
1600 PRINT
1700 PRINT "Type three numbers...": INPUT "(Put a comma between each
one)", A, B, C
1800 PRINT: PRINT "Those numbers are: "; A, B, C
1900 END
```

This example illustrates the various possibilities inherent in the INPUT statement. In line 1000, the comma at the end of promptString suppresses a question mark, whereas the semicolon at the end of promptString in line 1200 prints a question mark. However, in line 1300 the semicolon following INPUT suppresses the carriage return-line feed character at the end of the line. As a result, lines 1300 and 1400 print on the same line. See Figure 7-2 below.

Lines 1100 and 1200 combine a variable that gives information and one that asks for input. Finally, lines 1700 and 1800 show how to gather multiple items of information with one INPUT statement.



```
Please enter your first name  John
Okay, John, what is your last name?  Smith
Your area code?  415 And phone number?  961-4800
We can reach you at (415) 961-4800

Type three numbers...
(Put a comma between each one)  12.06,-3.14,+.00001

Those numbers are: 12.06      -3.14          0.00001
```

Figure 7-2. The INPUT Statement Illustrated

LOCATE

This statement positions the cursor to a specified dot or pixel location on the screen.

FORMAT

```
LOCATE x,y
```

NOTES

The horizontal coordinate (x) must be in the range of 1 to 320 and the vertical coordinate must be in the range of 1 to 240. The coordinates describe the position of the top, left pixel of the first character in the string that follows the LOCATE statement.

When a program runs, it doesn't normally display the cursor. When you follow a LOCATE statement with an input/output statement such as INPUT, the cursor appears at the screen location specified by the last preceding LOCATE statement. Similarly, a subsequent PRINT statement will output its data beginning at the previously specified screen location.

Also see the DRAWCHARS statement in Chapter Ten. DRAWCHARS does not position the cursor, but rather specific character strings.

EXAMPLE

```
1000 LOCATE 140,110
1100 INPUT "Horizontal axis (0-320)", Horiz
1200 LOCATE 140,120
1300 INPUT "Vertical axis (0-240)", Vert
1400 LOCATE Horiz, Vert
1500 PRINT "."
1600 END
```

This example shows the power of the LOCATE statement by positioning its INPUT statements (lines 1000 and 1200) and then by letting you display a dot at your own coordinates (line 1400).

PRINT

The PRINT statement displays data on the screen.

FORMAT

```
PRINT [expression][{,|;}][expression] ... [{,|;}]
```

NOTES

The PRINT statement displays any expression that follows it and sends a carriage return-line feed combination at the end of that expression. When deprived of an expression, PRINT displays a blank line, the result of the carriage return-line feed characters. The following expressions are all legal. Line 1100 yields the product of 5 x 6, 30.

```
    1000 PRINT "Hello"
    1100 PRINT 5*6
    1200 PRINT
```

You must enclose string constants in quotation marks ("). You can omit the final quotation mark from any string appearing at the end of a program line. Only the size of the screen limits the number of expressions a single PRINT statement can handle.

To place multiple expressions after a single PRINT statement, you must separate the individual expressions with either a comma (,) or a semicolon (;).

If you place a semicolon between two expressions, the two expressions will print with no intervening characters. See "SEMICOLON" later in this chapter.

If you place a comma between two expressions, PRINT displays the value of the second expression at the beginning of the next "print zone". GRiDBASIC divides each line into print zones of 15 spaces each. Commas used as expression separators cause a "tabbing" effect so that the next expression value is displayed in the next print zone.

The zones begin at columns 0, 15, 30, and 45. If a string has more than 15 characters, PRINT will skip the zone that has been overwritten and begin the next display at the next zone. Thus the comma never causes concatenation. See "COMMA" earlier in this chapter.

Terminating a list of expressions with a comma or semicolon, cancels the carriage return-line feed pair so that a subsequent PRINT statement continues printing on the same line. If a printed line is longer than the display's line width, printing continues on the next line. GRiDBASIC breaks strings at the right edge of the screen.

Printed numbers are always followed by one space and positive numbers are also preceded by one space. A minus sign precedes each negative number.

EXAMPLE

```
1000 LET A=5: B=3: C$="George": D$="Washington"
1100 PRINT A
1200 PRINT B
1300 PRINT "A+B=";A+B
1400 PRINT 5+3
1500 PRINT
1600 PRINT C$
1700 PRINT D$
1800 PRINT C$+D$
1900 PRINT
2000 PRINT TAB(0) "0"; TAB(10) "10"; TAB(20) "20"; TAB(30) "30";
TAB(40) "40"; TAB(50) "50"
2100 PRINT "A", "B", "C", "D", "E", "F", "G"
2200 PRINT "A"; "B"; "C"; "D"; "E"; "F"; "G"
2300 END
```

Lines 1100 and 1200 print the values stored at variables A and B. Line 1300 prints a string constant and then the result of adding A and B.

Line 1400 shows that PRINT can operate on numeric constants by doing math for you. Note in line 1800, a plus sign between string constants concatenates (or joins together) strings.

Line 2000 shows how the TAB statement operates with PRINT. Line 2100 the comma's tabbing effect and line 2200 the semicolon's concatenating effect.

PRINT USING

The PRINT USING statement formats strings or numbers, depending on the punctuation that follows the statement.

FORMAT

```
PRINT USING format symbol;{list of expression|list of string$}
```

NOTES

PRINT USING takes as its arguments a format symbol and a list either of numeric or string expressions. The format symbol shapes the expression into their format.

For example, the format symbol ("###.##")

```
     "###.##";A
```

tells GRiDBASIC to put the number stored in variable A into a format with three digits to the left of the decimal point and two digits to the right. Thus the number 34.14735 appears in the formatted form

```
      34.15
```

This section explores each format symbol and its results.

STRING EXPRESSIONS

You can modify string expressions with any one of three format symbols:

- The exclamation point (!)
- Double back slash enclosing space(s) (\n space\)
- The ampersand (&)

EXCLAMATION POINT (!)

The exclamation point returns only the first character in each string argument that follows it. See the example below.

DOUBLE BACK SLASH (\\)

When you don't put space(s) between the two back slash (CODE-SHIFT-') characters, the double back slash prints two characters from its string argument(s). Each space between the back slashes causes another character from the string(s) to print.

Double back slash prints one space character for each character you specify over the number of characters in the string. Thus if a string expression has five characters, and five space characters separate the two back slashes, two spaces will follow the printing of the five character string. See the example below with its companion printout.

AMPERSAND (&)

The ampersand causes the string to print exactly as it is stored. See the example below.

EXAMPLE (STRINGS)

```
1000 LET A$="Input"
1100 LET B$="Output"
1200 PRINT USING "!";A$;B$
1300 PRINT USING "\\";A$;B$
1400 PRINT USING "\ \";A$;B$
1500 PRINT USING "\  \";A$;B$
1600 PRINT USING "\   \";A$;B$
1700 PRINT USING "\    \";A$;B$
1800 PRINT USING "&";A$;
1900 END
```

```
IO
InOu
InpOut
InpuOutp
InputOutpu
Input Output
Input
```

NUMERIC EXPRESSIONS

The numeric format symbols include:

- The number or "pound" sign (#)
- The decimal point (.) and comma (,)
- The plus (+) and minus (-) signs
- The double asterisk (**)
- The double dollar ($$)
- The double asterisk-dollar (**$)
- The character string

NUMBER SIGN (#)

Each number sign reserves one digit of space for PRINT USING. Thus to reserve space for a five digit number followed by three decimal places, you write

```
     #####.###
```

Such a format handles numbers like

```
     12345.567 and -2345.987
```

Note that the minus sign takes one of the character positions. If you try to print a number with more digits than your format allows, a percent sign (%) will precede the first character (whether it's a sign or a number). We call this the "overflow symbol." See Figure 7-3 below. Thus trying to put the number -99999.01 in the format ######.## would result in

```
     %-99999.01
```

Whenever a number has fewer digits than the PRINT USING format, PRINT USING puts these extra spaces at the front of the number. Figure 7-3, shows a five digit format (to the left of the decimal) and three numbers in that format. Both the five digit positive number and the four digit negative number take up all alloted digits. The extra two digits pad the three digit number to its left.

```
#####.##
23468.91
  576.08
-3418.99
```

Figure 7-3. How Format Characters Pad Digits

COMMA (,)

If you want your number to display a comma every three digits, you can include the comma anywhere to the left of the decimal point. The comma also specifies another digit in the string. The following examples are all legal.

```
     #######,.##   ##,#####.##   ,#######.##
```

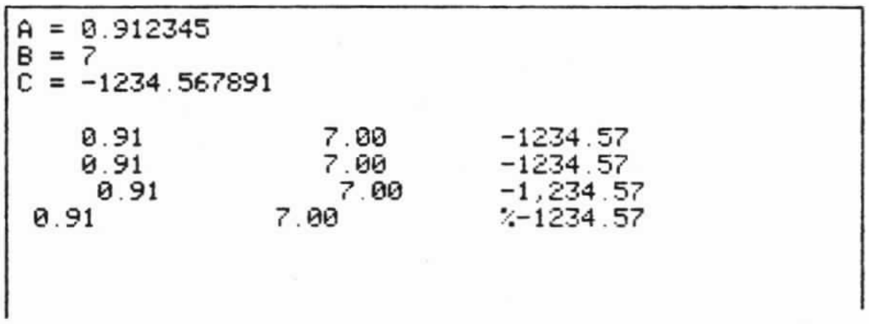
If you place a comma to the right of the decimal point, the comma prints as a literal at the end of the number. For example:

```
2345.56,
```

You can pad numeric output with surrounding spaces by putting space characters between either end of the string and the nearest quotation mark. See Figure 7.4 below for an example.

The program below with its output illustrate these facts.

```
1000 LET A=.912345
1100 LET B=7
1200 LET C=-1234.567891
1300 PRINT "A = ";A
1400 PRINT "B = ";B
1500 PRINT "C = ";C
1600 PRINT
1700 PRINT USING "#####.##";A, B, C
1800 PRINT USING "#####.##   ";A, B, C
1900 PRINT USING "#####,.##";A, B, C
2000 PRINT USING "##.##";A, B, C
2100 END
```



```
A = 0.912345
B = 7
C = -1234.567891

    0.91           7.00          -1234.57
    0.91           7.00          -1234.57
     0.91           7.00         -1,234.57
 0.91           7.00             %-1234.57
```

Figure 7-4 Basic formatting for PRINT USING

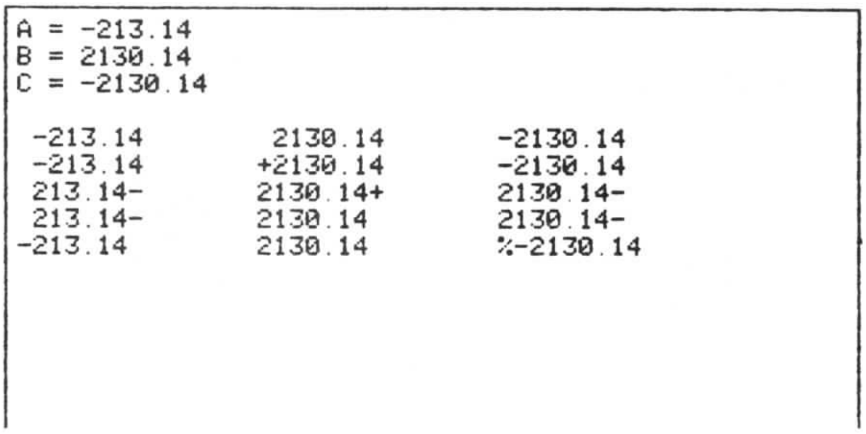
PLUS (+) AND MINUS (-) SIGNS

GRiDBASIC accepts a format with a plus or a minus sign at either the front or rear of the format string. All the following are legal:

```
+####.##
-####.##
####.##+
####.##-
```

Placing the plus sign on either end of the format string causes positive numbers to display the plus sign in the position indicated by the format. Negative numbers print the minus sign in this same position. Either sign adds an extra space to its number. See Figure 7-5 below, it shows a program with formatted output.

```
1000 LET A=-213.14
1100 LET B=2130.14
1200 LET C=-2130.14
1300 PRINT "A = ";A
1400 PRINT "B = ";B
1500 PRINT "C = ";C
1600 PRINT
1700 PRINT USING "#####.##";A, B, C
1800 PRINT USING "+####.##";A, B, C
1900 PRINT USING "####.##+";A, B, C
2000 PRINT USING "####.##-";A, B, C
2100 PRINT USING "-####.##";A, B, C
2200 END
```



```
A = -213.14
B = 2130.14
C = -2130.14

 -213.14          2130.14        -2130.14
 -213.14         +2130.14        -2130.14
 213.14-         2130.14+        2130.14-
 213.14-         2130.14         2130.14-
-213.14          2130.14         %-2130.14
```

Figure 7-5. The PRINT USING Format with Signs

DOUBLE ASTERISKS (**)

Placing two asterisks in front of the format string fills leading spaces with asterisks. NOTE: Leading spaces appear when the number of digits take less space than the number of positions specified by the format. The program and printout in Figure 7-6 illustrate double asterisk, double dollar, double asterisk-dollar formatting.

DOUBLE DOLLAR ($$)

Placing two dollar signs before a format string causes a dollar sign to print to the left of the formatted number. Double dollar creates two format spaces, one of which the printed dollar sign takes. See Figure 7-6 for examples.

DOUBLE ASTERISK-DOLLAR (**$)

Double asterisk-dollar combines the effects of double asterisk and double dollar: It prints a dollar sign to the left of the number and fills the field with asterisks whenever the number contains fewer digits than the format specifies. See Figure 7-6 immediately below.

EXAMPLE (NUMBERS)

```
1000 LET A=.912345
1100 LET B=7
1200 LET C=-1234.567891
1300 PRINT "A = ";A
1400 PRINT "B = ";B
1500 PRINT "C = ";C
1600 PRINT
1700 PRINT USING "$$######.##";A, B, C
1800 PRINT USING "$$#####.##-";A, B, C
1900 PRINT USING "**$####.##";A, B, C
2000 PRINT USING "**$###.##-";A, B, C
2100 PRINT USING "*#####.##";A, B, C
2200 END
```

```
A = 0.912345
B = 7
C = -1234.567891

    $0.91           $7.00      $-1234.57
   $0.91           $7.00       $1234.57-
*****$0.91     *****$7.00      **$-1234.57
****$0.91      ****$7.00       **$1234.57-
*****0.91      *****7.00       *-1234.57
```

Figure 7-6. Asterisk and Dollar Formatting

CHARACTER STRINGS

You can also include character strings between either set of quotation marks and the format string. Remember: The space is a character. All the following are legal:

```
     "  #####.##"
     "#####.##        "
     "   #####.##   "
     "Your account contains $$#####.##"
     "#####.## after deductions"
     "  You get #####.## shares for each hundred"
```

SEMICOLON

The semicolon character formats PRINT and INPUT data.

FORMAT

```
expression; expression[;]
```

NOTES

The semicolon character (;) serves to link expressions following PRINT and INPUT statements. Placed between expressions, the semicolon can link variables and strings. Unlike the comma, it provides no space between expressions. Placed at the end of a program line, the semicolon suppresses the carriage return-line feed characters issued by PRINT.

When a semicolon follows an INPUT statement, it suppresses the carriage return-line feed pair. As a result, you can request multiple items for INPUT on the same line.

EXAMPLE

```
1000 INPUT "Your first name is"; Name$
1100 PRINT
1200 INPUT; "Your City"; City$
1300 INPUT; " State"; St$
1400 INPUT " ZIP";ZIP$
1500 PRINT
1600 PRINT "Ahhh, you mean "; City$; ", "; St$; " "; ZIP$; " and not
";
1700 PRINT Name$;City$;St$;ZIP$
1800 END
```

In lines 1000, 1200, 1300, and 1400 the semicolons following the prompt string cause a question mark to print immediately after the prompt. The semicolons after INPUT on lines 1200 and 1300 suppress the carriage return-line feed pair, so that the program requests city, state, and ZIP code information all on the same line.

Line 1600 shows how you can place semicolons to link a mix of strings and variables. Line 1700 prints as one long line, because semicolons provide no spacing.

TAB

This function operates with the PRINT statement to tab horizontally a specified number of character positions or spaces.

FORMAT

```
TAB(expression)
```

NOTES

TAB understands expression as the column number where it should position whatever item follows it. For example,

```
1000 PRINT TAB(17) "Top Drawer"
```

prints the string "Top Drawer" at the 18th column of the current line. NOTE: The first display position on a line is 0; TAB(1) is the second character position on a line.

The expression must be a positive number. If the current print position is already beyond that specified by expression, TAB goes to the specified expression on the next line. If the specified value is greater than the length of a display line (52 characters), TAB simply keeps counting character positions on subsequent lines to arrive at the specified column position. Thus,

```
     PRINT TAB(52) "Here"
```

would print HERE beginning in the first character position of the next display line.
TAB operates only with the PRINT statement -- it does not work with the PRINT# statement.

EXAMPLE

```
1000 PRINT TAB(0) "0"; TAB(10) "10"; TAB(20) "20"; TAB(30) "30";
TAB(40) "40"; TAB(50) "50"
1100 FOR Position=0 TO 10
1200 PRINT TAB(Position) "Tab ";Position
1300 NEXT Position
1400 END
```

This program prints the word "Tab" and the tab's number at columns 0 through 10.

TIME$

The time function.

FORMAT

```
TIME$
```

NOTES

TIME$ returns the current time as a 13 character string from the Compass Computer system's real-time clock. The string takes the form hh:mm:ss a.m. or hh:mm:ss p.m. where hh is the hour (00 through 12), mm is the minutes (00 through 59) and ss is the seconds (00 through 59). NOTE: These characters are string, not numeric, characters. For a program to use them numerically, you must convert them to numbers (see Chapter Six, the VAL statement).

EXAMPLE

```
1000 PRINT "The time is "; TIME$
1100 FOR Loop=1 TO 5
1200 FOR Rest=1 TO 128: NEXT Rest
1300 LET Second$ = MID$(TIME$,7,2)
1400 PRINT: PRINT "The current seconds are "; Second$
1500 NEXT Loop
1600 END
```

After printing the time in line 1000, the program illustrates that you can take any particular element from the time string and work with it separately. In this case, the current seconds print every second, five times. The loop at line 1200 provides a one second (approximately) pause between printouts.

CHAPTER EIGHT: SEQUENTIAL FILES STATEMENTS

This chapter describes the statements necessary for writing, reading, and manipulating sequential files. Many of these commands also come into play when dealing with random access files (described in Chapter Nine).

The PRINT# and INPUT# statements, described in this chapter, transfer data to and from sequential files. The files created by the PRINT# statement are in a format called "interchange file format" and the INPUT# statement expects files it reads to be in this same format.

The interchange file format enables GRiD applications to place data in columns and rows for tabular or cell-based applications such as GRiDPLAN, GRiDFILE and GRiDPLOT. Because GRiD applications can read files in interchange format, they can process data generated by GRiDBASIC programs.

NOTE: Many of the examples in this book write or read data from a floppy drive and a subject called "'Testing'." For example:

```
     1000 OPEN "O",1,"'f0'Testing'Weekly"
```

If you prefer to put your test files on your bubble, replace "'f0" with "'b0." For the hard disk, substitute "'w0." The GRiDBASIC OPEN command can create a title (like "'Weekly") but not a subject. If you want to create a special subject for your examples and test programs, you must do this beforehand.

If you prefer not to deal with pathname syntax, put the GETFILE$ statement in your program. It presents the standard application file form. See the GETFILE$ statement later in this chapter.

NOTE: You need only specify a file's device, subject, and/or kind when any of those designations change. For example, if you set usage (CODE-U) to the floppy drive and then select the subject "'Testing" to write a BASIC program in, you could write

```
     1000 OPEN "O",1,"'Weekly"
```

instead of

```
     1000 OPEN "O",1,"'f0'Testing'Weekly~Text"
```

Like GRiDWRITE, GRiDBASIC assigns "Text" as the kind for file's created with the OPEN and PRINT# statements.

CLOSE

This statement closes a file or files previously enabled for program access by an OPEN statement.

FORMAT

```
CLOSE [[#]fileTag][,[#]fileTag][, ... ]
```

NOTES

The OPEN statement assigns a file tag number to a particular file name. The CLOSE statement disassociates the tag from this file name, so that you can reassign it to another file. With sequential files, you must close a file to change its mode.

For example, you have to close a program operating a file in the output mode before you can append to it. You can reopen the same file again with its previous tag or a different tag. For a further discussion of fileTags and modes, refer to the OPEN statement later in this chapter.

If you fail to give the CLOSE statement a fileTag, GRiDBASIC closes all open files. NOTE: An END statement automatically closes all files, but a STOP statement does not. GRiDBASIC allows the optional number sign (#) that precedes the fileTag to provide compatibility with other versions of BASIC.

EXAMPLE

```
1000 OPEN "I",1,"‘f0‘Testing‘Weekly~"
1100 WHILE NOT EOF(1)
1200 INPUT# 1, Day$
1300 PRINT Day$
1400 WEND
1500 CLOSE 1
1600 PRINT: PRINT "Those are the days of our lives."
1700 END
```

In this example (taken from the OPEN statement below) line 1500 closes the "‘Testing" file opened in line 1000. You can also close multiple files with the same statement for example:

```
     1000 CLOSE 3,4,15
```

EOF

The end of file function.

FORMAT

```
EOF(fileTag)
```

NOTES

The EOF function returns a value that indicates if an end of file has been reached on a specified file. If the end of file has been reached, EOF returns a -1 (true) value. If the end of file has not been reached, a 0 (false) is returned.
The fileTag parameter is the number you specified when you opened the file for input.

EXAMPLE

```
1000 OPEN "I",1,"`f0`Testing`Weekly~"
1100 WHILE NOT EOF(1)
1200 INPUT# 1, Day$
1300 PRINT Day$
1400 WEND
1500 CLOSE 1
1600 PRINT: PRINT "Those are the days of our lives."
1700 END
```

NOTE: This example is one of three illustrating the OPEN command. To make this work program work, you will have to type and run the OPEN Output example first (see the OPEN statement later in this chapter).

EOLN

The end of line function.

FORMAT

```
EOLN(fileTag)
```

NOTES

The EOLN function returns a value that indicates if an end of line within a specified file has been reached. An end of a line within a file is indicated by the carriage return-line feed combination. If the most recent character read from a file is followed by a carriage return-line feed, or if the end of file has been reached, the EOLN function returns a -1 (true) value; otherwise, it returns a 0 (zero).

This function is especially useful when reading interchange files from other GRiD applications.

The fileTag parameter is the number you specified when the file was opened for input.

EXAMPLE

```
1000 OPEN "I",1,"`f0`Testing`AllChecks~MyKind"
1100 INPUT "Take balance from what row (2-13)"; Row
1200 LET Lines=0
1300 WHILE NOT EOF(1)
1400      IF EOLN(1) THEN LET Lines=Lines+1
1500      INPUT# 1, Record$
1600      IF Lines = Row THEN LET Goal$ = Record$
1700 WEND
1800 PRINT
1900 PRINT "The balance at row ";Row;" is ";Goal$
2000 END
```

This program lets you take any number from the balance column of the worksheet shown below in Figure 8-1. For example, if you select row 5, the program will return the amount 479251.43 (the amount after the 02/03 Deposit).

If you want to set up Figure 8-1 in a worksheet, make all but the first item in the balance column a formula that adds current line's Amount to the previous Balance column amount. The first item is the absolute amount, 491084.00. The second Balance item (478605.47) results from adding -12478.53 to the absolute amount.

Check no	Payee	Amount	Balance
	start balance		491084.00
1000	Acme Realty	-12478.53	478605.47
1001	Local Power	-5601.89	473003.58
1002	Telephone	-3016.92	469986.66
23457	02/03 Deposit	9264.77	479251.43
1003	Fass Freight	-1032.14	478219.29
1004	Ace Credit	-15629.01	462590.28
1005	Fleet Rents	-4912.30	457677.98
1006	Personnel	-35971.95	421706.03
1007	A-1 Cleaning	-856.75	420849.28
1008	StarInsurance	-1478.42	419370.86
1009	Heavy Equip	-25819.66	393551.20

Figure 8-1. Worksheet Figures for Example Program

The EOLN function works by searching for the carriage return-line feed combination. In the case of this example program, line 1400 increments a line counter (the variable "Line") each time it encounters an EOLN. When the value of Line equals the value of Row (input by the user), the program prints the last field.

NOTE: Record$ reads one record at a time, not one line. One cell, begun and/or ended by the Tab character constitutes a record.

GETFILE$

The get file statement.

FORMAT

```
string$=GETFILE$("promptMessage")
```

NOTES

Normally, programmers specify file pathnames with the program development syntax --

```
`Device`Subject`Title~Kind~
```

For example: 1000 OPEN "I",1,"`f0`Testing`Weekly~Text~"

The GETFILE$ statement lets you bypass this syntax by bringing you the standard file form. Filling in the form and confirming it brings you the desired file.

You may prefer GETFILE$ over the pathname syntax if you have trouble understanding pathname syntax or if you want your program to work with different files. On the other hand, if your program uses just one file (or only a few -- you could change a parameter before running the program), go with pathname syntax. Likewise, if you value quick access time, choose pathname syntax.

EXAMPLE

```
1000 MyFile$=GETFILE$("Select file and confirm")
1100 OPEN "I",1, MyFile$
1200 WHILE NOT EOF(1)
1300     INPUT# 1, Day$
1400     PRINT Day$
1500 WEND
1600 END
```

In this example, line 1000 assigns the GETFILE$ function to the string variable, MyFile$ along with the prompt

```
Select file and confirm
```

The prompt appears in the message line when you run the program. Once you give the file information to the form and confirm, the string variable delivers that information to the program (see line 1100).

INPUT#

This statement assigns values to program variables by reading data items from a sequential file.

FORMAT

```
INPUT# fileTag,variablesList
```

NOTES

The fileTag parameter is the number you specified when the file was opened for input.

Data items read from the file are assigned to the variables specified in the variablesList. Each data item read from the file must be of the same type as that specified by the corresponding variable name. The variable names in variablesList can be any mix of numeric and string variable names, including subscripted variables.

INPUT# expects the file to be in GRiD's standard interchange file format: data items are separated by Horizontal Tabs or Carriage Return-Line Feed pairs. The PRINT# creates this interchange file format, as do GRiD's cell-based applications such as GRiDFILE and GRiDPLAN.

If the end of a file is reached while an item is being input, the item is terminated. If a type mismatch occurs between the data item and the variable that it is being assigned to, or the file has an insufficient number of items, the program halts and an appropriate error message appears.

The INPUT# statement can obtain data from the keyboard; you open the keyboard just as you would any other file. The keyboard's filename is "CI" (for Console Input). For example,

```
OPEN "I",1,"CI"
```

If you choose keyboard input, you must write a prompt for your user(s); unlike the INPUT statement, the INPUT# statement does not print a question mark or a prompt message.

EXAMPLE

```
1000 OPEN "I",1,"`f0`Testing`Weekly~"
1100 WHILE NOT EOF(1)
1200 INPUT# 1, Day$
1300 PRINT Day$
1400 WEND
1500 CLOSE 1
1600 PRINT: PRINT "Those are the days of our lives."
1700 END
```

NOTE: This example is one of three illustrating the OPEN command. To make this work program work, you will have to enter and run the OPEN Output example first.

INPUT$

Prefer INPUT$ over INPUT# for handling communications files or for reading large sections of files.

FORMAT

```
INPUT$(tag#, bytes)
```

NOTES

INPUT$ fetches the number bytes (or characters) assigned to it in its argument from the file represented by the file tag number. You can assign part or all of the characters read from a communications or other file into one string with INPUT$.

NOTE: If you give the statement a greater number of characters to fetch than exist within the file, INPUT$ quits when it reaches the end of file character.

EXAMPLE

```
1000 OPEN "I",1, "‘f0‘Testing‘AnotherDay~"
1100 INPUT "Get how many characters from this file"; HowMany
1200 WantToSee$=INPUT$(1,HowMany)
1300 PRINT
1400 PRINT WantToSee$
1500 END
```

This example has the INPUT$ statement fetch as many character from the file of weekdays as the user specifies. Unlike INPUT#, INPUT$ does not convert the end-of-line characters (carriage rerturn-line feed). Rather, it prints the entire string of characters without breaking at the end of lines (except for the right margin).

NOTE: The OPEN output example creates a text file with the days of the week in it. You can create the same file by invoking GRiDWRITE and typing the days in a vertical list. The INPUT$ program above can read it, as it can read any text file.

KILL

The KILL statement erases a file.

FORMAT

```
KILL filename
```

NOTES

Follow the KILL statement with the file name of the file you want to erase. You can present this in the form of a string variable. In fact, the most efficient way to issue a KILL is with a file form created with the GETFILE$ statement (discussed earlier in this chapter). GETFILE$ delivers its data to a string variable.

EXAMPLE

```
1000 OPEN "O",1,"'f0'Testing'NewFile": CLOSE 1
1100 PRINT "NewFile created!"
1200 PRINT "KILL NewFile by selecting it."
1300 LET Joy$=GETFILE$ ("Select FloppyDisk-Testing-NewFile-Text and
confirm")
1400 KILL Joy$
1500 PRINT: PRINT "NewFile KILLed.  See if NewFile is still there."
1600 Search$=GETFILE$ ("Press ESC after viewing files")
1700 PRINT: PRINT "KILL erased the file."
1800 END
```

Line 1000 creates a file. In line 1300, the GETFILE$ statement presents a file form. We recommend GETFILE$ over typing file name syntax. Line 1400 erases the file named in the form. Line 1600 presents second file form, so that you can see for yourself that KILL indeed erased the file.

LOC

The locating statement.

FORMAT

```
expression=LOC(tag#)
```

NOTES

LOC locates a portion of a file by returning a number from a file. What that number represents depends on the type of file involved.

Random	The record number of the last record read or written.
Sequential	The number of records read or written since the last OPEN.
Communications	The number of characters waiting to be read in the input buffer.

EXAMPLE

```
1000 OPEN "I",1,"`f0`Testing`AnotherDay~"
1100 PRINT "Record","Byte":PRINT
1200 WHILE NOT EOF(1)
1300      LET MyByte=LOC(1)
1400      INPUT# 1, Day$
1500      PRINT Day$, MyByte
1600 WEND
1700 CLOSE 1
1800 END
```

This example reads a sequential file. This example gets the byte number of each record in a days-of-the-week file. If you want to run this file, you can create a text file called "`AnotherDay" by typing in the days of the week in GRiDWRITE, putting each day on its own line.

When you run this example, the "S" beginning Sunday appears as the byte 1. The "M" in Monday as the ninth character. Why? Because in addition to the six characters in "Sunday," LOC also counts the two invisible characters at the end of the line -- carriage return and line feed. Remember: LOC returns the absolute position of each byte.

LOF

The length of file statement.

FORMAT

```
expression=LOF(fileTag)
```

NOTES

LOF returns a file's length in bytes. You must supply the file's file tag in paratheses.

EXAMPLE

```
1000 OPEN "I",1, "‘f0‘Testing‘AnotherDay~"
1100 Length=LOF(1)
1200 PRINT
1300 PRINT "The length of this file is "; Length; " characters."
1400 END
```

OPEN

This statement opens a file for a particular kind of access.

FORMAT

```
OPEN "accessMode"[#]fileTag,"fileName"
```

NOTES

Here is a typical OPEN statement:

```
1000 OPEN "I", 3, "`w0`Taxes`January"
```

The accessMode parameter specifies the way that subsequent PRINT# and INPUT# statements in a program can access this file. Further, we can make subsequent references to this program with the number three, instead of with the pathname "`w0`Taxes`January." GRiDBASIC has three access modes:

"I"	specifies sequential input
"O"	specifies sequential output
"A"	specifies sequential output to be appended

Note that a single OPEN statement can only establish access for one sequential activity at a time. A sequential file cannot be OPEN for both input and output at the same time. To change the type of access you have assigned to a file, you must first CLOSE the file, then execute another OPEN statement specifying the new type of access.

The fileTag parameter is a number that you specify to be associated with this fileName for a particular OPEN operation. Subsequent accesses to the file with PRINT# or INPUT# statements can then refer to the file simply by the fileTag number; you need not specify the file name or type of access. NOTE: GRiDBASIC allows the optional number sign (#) that precedes the fileTag for compatibility with other versions of BASIC.

A file can be open under only one fileTag number at a time. You cannot have a file simultaneously OPEN for input and output, for multiple inputs, or for multiple outputs. To perform two access operations, you need two open operations and two tag numbers. For example:

```
1000 OPEN "I",1 "`MyFile"
1100 OPEN "O",2 "`YourFile"
```

This opens the file titled "`MyFile" for input and gives it tag number 1. Line 1100 opens a second file to receive this data (output), "`YourFile," with the tag number 2.

The fileName parameter can be any name you have specified up to 80 characters in length. For details, see "File Naming Conventions" near the end of Chapter 2. You can also use the standard Compass file form to get file names for your BASIC programs; see the GETFILE$ statement earlier in this chapter.

EXAMPLE (OUTPUT)

```
1000 OPEN "O",1,"`f0`Testing`Weekly~"
1100 DATA Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday
1200 FOR Week=1 TO 7
1300 READ Day$
1400 PRINT Day$
1500 PRINT# 1, Day$
1600 NEXT Week
1700 CLOSE 1
1800 PRINT: PRINT "The Weekly file is closed."
1900 END
```

This example creates a file with the title "`Weekly" and writes the names of the days of the week into it.

EXAMPLE (INPUT)

```
1000 OPEN "I",1,"`f0`Testing`Weekly~"
1100 WHILE NOT EOF(1)
1200 INPUT# 1, Day$
1300 PRINT Day$
1400 WEND
1500 CLOSE 1
1600 PRINT: PRINT "Those are the days of our lives."
1700 END
```

This example opens the previous file, retrieves the days of the week from it, and prints them on the screen.

EXAMPLE (APPEND)

```
1000 OPEN "A",1,"`f0`Testing`Weekly~"
1100 DATA Yesterday, Today, Tomorrow, The Day after that
1200 FOR Now = 1 TO 4
1300 READ Day$
1400 PRINT# 1, Day$
1500 PRINT Day$
1600 NEXT Now
1700 CLOSE 1
1800 PRINT: PRINT "Those were the days, my friend."
1900 END
```

This example appends four lines to the original file -- Yesterday, Today, Tomorrow, and The Day after that. If you re-run the INPUT example, you will see these new additions.

PRINT#

The PRINT# statement writes data to a sequential file.

FORMAT

```
PRINT#fileTag,expression[{,|;}][expression] ... [{,|;}]
```

NOTES

The fileTag parameter is the number you specified when you opened the file for output. It identifies the sequential file that is to receive the data.

PRINT# writes the data contained in the expession(s) to the file with appropriate delimiting characters automatically inserted. Your choice of punctuation (either a comma or a semicolon) between expressions determines the delimiting characters written to the file to separate the items in each expression. You can use either commas (,) or semicolons (;) as separators.

NOTE: If you intend to print to an Epson printer and want your commas to perform a tabbing function, see the "Epson Notes" at the end of this discussion.

If you place a semicolon between two expressions, PRINT# writes the values of the two expressions with no delimiting character between them.

If you place a comma between two expressions, a horizontal tab character is written to the file separating the contents of the first expression from the contents of the second expression.

If a list of expressions terminates without a comma or semicolon, PRINT# writes a carriage return-line feed at the end of the list. If a comma terminates a list of expressions, PRINT# places a horizontal tab character after the last expression. If a semicolon terminates a list of expressions, it suppresses any delimiting character. Thus a subsequent PRINT# statement begins writing data to the file beginning at the point where the last PRINT# left off.

NOTE: The format of the file created by the PRINT# statement is compatible with the interchange file format. As a result, cell-based GRiD applications such as GRiDPLOT, GRiDFILE and GRiDPLAN can work with these files.

Choose the INPUT# statement to input data from a file that you created with the PRINT# statement.

EPSON NOTES

For the Epson to interpret GRiDBASIC's commas correctly -- providing tabs -- you must follow the PRINT# command with the file tag number, an ESC D (represented by CHR$(27)+"D") and the column number of each tab preceded by the CHR$ statement. Concatenate these tab positions with the plus sign (+). All such statements must end with the null character (CHR$(0). Do NOT exceed an 80-character line. An example command assigning 15 character-wide tabs follows:

```
PRINT# 1, CHR$(27)+"D"+CHR$(15)+CHR$(30)+...+CHR$(0)
```

EXAMPLE

```
1000 OPEN "O",1,"'f0'Testing'Weekly~"
1100 DATA Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday
1200 FOR Week=1 TO 7
1300 READ Day$
1400 PRINT Day$
1500 PRINT# 1, Day$
1600 NEXT Week
1700 CLOSE 1
1800 PRINT: PRINT "The Weekly file is closed."
1900 END
```

This example (from the OPEN statement) writes the days of the weeks to a file called "'Weekly." The PRINT# statement in line 1500 transmits this data one string (or day!) at a time.

PRINT# USING

The PRINT# USING statement writes data to sequential files or to the printer in a specified format.

FORMAT

```
PRINT# fileTag, USING; formatstring; {numericvar|stringvar list}
```

NOTES

PRINT# USING takes the same arguments as PRINT USING. The only syntactical difference between the two is the presence of the number sign (#) and tag number following the word "PRINT." For details on this statement's many formatting possibilities, see PRINT USING in Chapter Seven.

EXAMPLE

```
1000 OPEN "O",1, "*epson"
1100 FOR Times = 1 TO 12
1200 LET Number = 100000*RND(1)
1300 PRINT Number;"    ";: PRINT USING "$$#######,.##"; Number
1400 PRINT# 1, Number;
1500 PRINT# 1, USING "    $$#######,.##";Number
1600 NEXT Times
1700 CLOSE 1
1800 END
```

This example generates 12 random numbers and then prints and formats them -- to the screen and to the printer. Line 1400 prints the unformatted number; line 1500 does the formatting. In this case, we have turned the number into a dollar amount and preceded that with spaces to separate the unformatted and formatted numbers. Figure 8-2 below is a typical printout.

Note that to print, you must open your printer as a file (line 1000) and give it a tag number. The PRINT#, PRINT# USING and CLOSE statements all take advantage of this tag number.

This example creates a printout to the screen and on paper that resembles the printout below in Figure 8-2. The left column displays the random number we generated. The right column shows how PRINT# USING formatted the same number.

```
6001.3733119707      $6,001.37
62092.0119020371     $62,092.01
38194.8577096208     $38,194.86
82920.5767910277     $82,920.58
7277.02754253452      $7,277.03
81165.7892729076     $81,165.79
3288.31921873808      $3,288.32
88273.4416723888     $88,273.44
89706.2638284886     $89,706.26
52875.5626764324     $52,875.56
65941.8631265736     $65,941.86
53293.6598764019     $53,293.66
```

Figure 8-2. PRINT# USING formatting of Random Numbers

CHAPTER NINE: RANDOM FILE STATEMENTS

Random access (also called "direct access") files differ from sequential files in several important ways. First, each data unit or record is of a fixed length (specified by the programmer). Second, you can go directly to any record within a random access file, rather than having to go through the entire file. This is true for both read and write activities. Third, random files have a buffer in RAM memory. Your program interacts with the buffer, rather than directly with your storage device.

Random access files share statements with sequential files. Note, however, that some of these statements don't behave exactly the same. For example, the LOF statement in a sequential file returns the length of that file in bytes. In a random file, LOF returns the total number of records in the file. These two numbers only equal each other when a random file has one-byte long records!

Another example. The OPEN statement has only one access mode for opening a random access file -- "R" (Sequential files have three). With random files, it doesn't matter whether you are opening the file for INPUT or OUTPUT. Further, the OPEN statement in a random access file also takes an optional argument after the file name, the buffer length.

Although this chapter contains plenty of working examples, you may want to look at the basic steps involved in creating random access write and read files. First to create a random access file and write data to it, follow these steps.

▶ A RANDOM ACCESS WRITE FILE

① OPEN the file with an "R" and an optional buffer size specification.

```
1000 OPEN "R",1,"`f0`Testing`Demograf.1~",30
```

② Define the sizes of the fields in your record buffer with the FIELD statement.

```
1100 FIELD 1, 12 AS Name$, 3 AS Age$, 15 AS City$
```

③ Gather the data you want to write. You can do this by reading from other files, with INPUT and/or READ DATA statements, for example.

```
1200 INPUT "Name"; N$
```

④ Put this data into the buffer with the LSET or RSET statements. Use the variable names you assigned to fields in the FIELD statement.

```
1600 LSET Name$=N$
```

⑤ Write the data to the file with the PUT statement.

```
2000 PUT 1
```

⑥ Close the file.

```
2200 CLOSE 1
```

Here's an outline for reading data from a file.

▶ A RANDOM ACCESS READ FILE

① OPEN the file with an "R" and an optional buffer size specification.

```
1000 OPEN "R",1,"`f0`Testing`Demograf.1~",30
```

② Define record buffer field sizes in with the FIELD statement.

```
1100 FIELD 1, 12 AS Name$, 3 AS Age$, 15 AS City$
```

③ Use the GET statement to read your data from the file.

```
1500 GET 1, RecordNo
```

④ Process this data and send it to the screen or other device.

```
1600 PRINT Name$;
```

⑤ Close the file.

```
2400 CLOSE 1
```

CVI, CVS, CVD

The convert string to integer function.
The convert string to single-precision function.
The convert string to double-precision function.

FORMAT

```
CVI(2-byte string)
CVS(8-byte string)
CVD(16-byte string)
```

NOTES

Programmers often convert numeric values to strings so they can format these values with the LSET or RSET statements. However, the system cannot perform mathematical operations on string values. Only on numeric values. Therefore, they convert string values back to numbers. CVI converts a 2-byte string, CVS converts a 4-byte, and CVD converts an 8-byte string.

Choose the CV function that matches the MK$ function that made the original number into a string. Table 9-1 below illustrates this.

MK$ Form	CV Form	No. of Bytes
MKI$	CVI	2
MKS$	CVS	4
MKD$	CVD	8

Table 9-1. Choosing MK$ and CV Functions

EXAMPLE

```
1000 OPEN "R",1,"'f0'Testing'MyNumbers~",8
1100 FIELD 1, 8 AS Number$
1200 FOR Count=1 TO 3
1300 INPUT "Any number"; N
1400 LSET Number$=MKD$(N)
1500 PUT 1
1600 NEXT Count
1700 PRINT:PRINT "MKD$ Form"; TAB(13) "After CVD": PRINT
1800 FOR Count=1 TO 3
1900 GET 1, Count
2000 PRINT Number$;
2100 PRINT TAB(13) CVD(Number$)
2200 NEXT Count
2300 PRINT
2400 END
```

This program asks you to enter any three numbers. It then converts them to string format in line 1400. It then writes these numbers t the screen and shows them in the form in which they are stored (the "MKD$ Form") and the numeric form they take after conversion to double precision (CVD). See the MKD$ function below for details on its operation.

FIELD

FIELD sets up a random file buffer.

FORMAT

```
FIELD [#] tag#, number AS string$ [, number AS string$] ...
```

NOTES

The FIELD statement breaks the buffer into individual fields. Thus the buffer is the length of the record that comprises these fields. To maximize efficient use of memory and storage space, add the numbers of characters for each field together and give the resulting sum as the optional buffer length parameter.

The AS statement assigns buffer space in characters (indicated by the number preceding AS) to a variable (following AS).

EXAMPLE

```
1000 OPEN "R",1,"‘f0‘Testing‘Demograf.1~",30
1100 FIELD 1, 12 AS Name$, 3 AS Age$, 15 AS City$
1200 WHILE NOT EOF(1)
1300 GET 1
1400 PRINT Name$;
1500 PRINT Age$;
1600 PRINT City$
1700 PRINT
1800 WEND
1900 CLOSE 1
2000 END
```

This program reads in all the records from the ‘Demograf.1 file. It allots space in the (random) buffer for its three fields as follows: 12 characters for the string variable Name$, 3 characters for the string variable AGE$, and 9 to the string variable ZIP$.

GET

The GET statement retrieves data for random file access.

FORMAT

```
GET [#] tag#[, number]
```

NOTES

The GET statement reads one record at a time into the buffer. If you do not specify a number, any reading of these records causes their content to appear in the order in which they exist in the file. If you specify a number, the record belonging to that record number appears. Thus if you ask for

```
1500 GET 1,3
```

line 1500 will get the third record from the file you assigned the tag number of 1.

EXAMPLE

```
1000 OPEN "R",1,"`f0`Testing`Demograf.1~",30
1100 FIELD 1, 12 AS Name$, 3 AS Age$, 15 AS City$
1200 LET Items=LOF(1)
1300 PRINT "Record number (between 1 and "; Items;
1400 INPUT ") please", RecordNo
1500 GET 1, RecordNo
1600 PRINT Name$;
1700 PRINT Age$;
1800 PRINT City$
1900 INPUT "This person's true age", A$
2000 PRINT
2100 LSET Age$=A$
2200 PUT 1, RecordNo
2300 GOTO 1300
2400 CLOSE 1
2500 END
```

This example reads whatever record number you specify (in line 1400) and prints the appropriate data on the screen. The program then gives you the opportunity to change the age parameter.

LOC

LOC locates a record.

FORMAT

LOC(tag#)

NOTES

LOC returns the record number of the next record that you can either GET or PUT. When this function sees the EOF marker, it looks no further.

EXAMPLE

```
1000 OPEN "R",1,"‘f0‘Testing‘Demograf.1~",30
1100 FIELD 1, 12 AS Name$, 3 AS Age$, 15 AS City$
1200 WHILE LOC(1)<=8
1300 PRINT LOC(1);"  ";
1400 GET 1
1500 PRINT Name$;
1600 PRINT Age$;
1700 PRINT City$
1800 PRINT
1900 WEND
2000 CLOSE 1
2100 END
```

This example uses LOC to test whether the WHILE WEND should continue (line 1200) and to print each record's number before printing the contents of the record (line 1300).

LOF

The length of file statement

FORMAT

```
LOF(fileTag)
```

NOTES

LOF returns a file's length in records.

EXAMPLE

```
1000 OPEN "R",1,"`f0`Testing`Demograf.1~",30
1100 FIELD 1, 12 AS Name$, 3 AS Age$, 15 AS City$
1200 WHILE LOC(1) <= LOF(1)
1300 PRINT LOC(1);"  ";
1400 GET 1
1500 PRINT Name$;
1600 PRINT Age$;
1700 PRINT City$
1800 PRINT
1900 WEND
2000 CLOSE 1
2100 END
```

In this example, line 1200 we test whether to continue the WHILE WEND loop by comparing the file record number (LOC) with the number of records in the file (LOF). If the record number is less than or equal to LOF, the WHILE WEND loop continues.

LSET and RSET

The LSET and RSET statements

FORMAT

```
LSET fieldString=programString
RSET fieldString=programString
```

NOTES

LSET and RSET statements assign a string created within the current program to one of the string variables defined in the FIELD statement. In the event that a value does not take up all the string space allotted to it, LSET will left-justify the value within the space. Similarly, RSET right-justifies when space remains. NOTE: You must convert numeric variables to string variables before doing this. Either the MK$ statement (see below) or the STR$ can do the job.

CAUTION: Do not use a field variable in an input statement nor put it on the left side of an assignment (LET) statement. Either practice causes the variable pointer to point not to the random file buffer, but to string space. The result: garbage in your file.

EXAMPLE

```
1000 OPEN "R",1,"'f0'Testing'Demograf.1~",30
1100 FIELD 1, 12 AS Name$, 3 AS Age$, 15 AS City$
1200 INPUT "Name"; N$
1300 IF N$="=" THEN GOTO 2200
1400 INPUT "Age"; A$
1500 INPUT "City"; C$
1600 LSET Name$=N$
1700 LSET Age$=A$
1800 LSET City$=C$
1900 PRINT
2000 PUT 1
2100 GOTO 1200
2200 CLOSE 1
2300 PRINT: PRINT "This input session is over"
2400 END
```

This example writess data to a random access file. The LSET statements (lines 1600-1800), it assign the values in the input variables (lines 1200, 1400, and 1500) to the field variables (assigned in line 1100). If you wanted your output right-justified instead of left-justified, you would substitute RSET for each occurence of LSET.

MKI$, MKS$, MKD$

The make string function.

FORMAT

```
MKI$ (expression)
MKS$ (expression)
MKD$ (expression)
```

NOTES

The MK$ function converts numeric expressions (including variables and numbers) into 4-byte strings. You must convert any numeric expressions before submitting them to the LSET or RSET. (You must choose one of these two to put data into the buffer).

As a general rule, choose MKD$ to convert your strings. At 8 bytes, it yields the greatest precision and, with its corollary CVD, minimizes the possibility for returning an inaccurate number from storage.

EXAMPLE

```
1000 OPEN "R",1,"`f0`Testing`MyNumbers~",8
1100 FIELD 1, 8 AS Number$
1200 FOR Count=1 TO 3
1300 INPUT "Any number"; N
1400 LSET Number$=MKD$(N)
1500 PUT 1
1600 NEXT Count
1700 PRINT:PRINT "MKD$ Form"; TAB(13) "After CVD": PRINT
1800 FOR Count=1 TO 3
1900 GET 1, Count
2000 PRINT Number$;
2100 PRINT TAB(13) CVD(Number$)
2200 NEXT Count
2300 PRINT
2400 END
```

This program asks you to enter any three numbers. It then converts them to string format in line 1400. Note that we put the result of our MKD$ function in the string variable, Number$. Lines 1900 to 2100 bring the numbers back from storage -- in the MK$ string form and in the converted form. See the CVD function above for details on its use.

OPEN

The OPEN statement creates a buffer in RAM memory and prepares the system to write data to or read data from the specified file. If the file doesn't exist, OPEN creates the file title. It cannot, however, create a subject.

FORMAT

```
OPEN "R"[#]fileTag,"fileName",[bufferLength]
```

NOTES

Here is a typical OPEN statement:

```
     1000 OPEN "R", 3, "`w0`Taxes`January", 63
```

When placed in the context of random access files, OPEN has only one accessMode parameter -- "R" for Random. This specifies the kind of file manipulation activities the fill allows. NOTE: where sequential files have three possible letters -- "I," "O," and "A," random files have just one accessmode -- "R."

NOTE: Unlike sequential files that must CLOSE and issue a new OPEN statement before changing its access activity, random files can do both input (GET) and output (PUT) under the same OPEN statement. See the example under the MKD$ command. For details on sequential files, see the OPEN statement in Chapter Eight.

The fileTag parameter is a number that you specify to be associated with this fileName for a particular OPEN operation. Subsequent accesses of the file with GET or PUT statements can then refer to the file simply by the fileTag number; you need not specify the file name or type of access. In the example above, this number is 3.

A file can be open under only one fileTag number at a time. NOTE: GRiDBASIC allows the optional number sign (#) preceding fileTag for compatibility with other versions of BASIC.

The fileName parameter can be any name you have specified up to 80 characters in length. For details, see "File Naming Conventions" near the end of Chapter 2.

The optional bufferLength parameter sets the size of the buffer. For greatest efficiency, you should assign this the same number of bytes as the total number bytes in the field statement. In the example above, we defined the length of the buffer as 63 characters. The default length for the buffer is 128 bytes.

EXAMPLE

```
1000 OPEN "R",1,"‘f0‘Testing‘Demograf.1~",30
1100 FIELD 1, 12 AS Name$, 3 AS Age$, 15 AS City$
1200 INPUT "Name"; N$
1300 IF N$="=" THEN GOTO 2200
1400 INPUT "Age"; A$
1500 INPUT "City"; C$
1600 LSET Name$=N$
1700 LSET Age$=A$
1800 LSET City$=C$
1900 PRINT
2000 PUT 1
2100 GOTO 1200
2200 CLOSE 1
2300 PRINT: PRINT "This input session is over"
2400 END
```

This example inputs data to a random access file. Its OPEN statement assigns this file the access mode parameter, "R" and file tag number "1." It then specifies the floppy drive as the device, "Testing" as the subject, and "Demograf.1" as the title. Finally, it sets aside 30 bytes for the file's buffer length. Note that 30 is the sum of the lengths of the three records given in the FIELD statment (line 1100).

PUT

The PUT statement writes data to a random buffer.

FORMAT

```
PUT [#]fileTag [, expression]
```

NOTES

The PUT statement writes data to a random file buffer for transfer to the appropriate storage medium. PUT understands the optional expression (whether a constant or a variable) as a record number. If another record already has the number you specify, PUT will write over it. If you fail to specify a number, PUT assigns the next available number.

EXAMPLE

```
1000 OPEN "R",1,"'f0'Testing'Demograf.1~",30
1100 FIELD 1, 12 AS Name$, 3 AS Age$, 15 AS City$
1200 INPUT "Name"; N$
1300 IF N$="=" THEN GOTO 2200
1400 INPUT "Age"; A$
1500 INPUT "City"; C$
1600 LSET Name$=N$
1700 LSET Age$=A$
1800 LSET City$=C$
1900 PRINT
2000 PUT 1
2100 GOTO 1200
2200 CLOSE 1
2300 PRINT: PRINT "This input session is over"
2400 END
```

This program puts data into a file called 'Demograf.1. Each time through the loop, it puts a record with three fields -- Name$, Age$ and City$.

CHAPTER TEN: GRAPHICS STATEMENTS

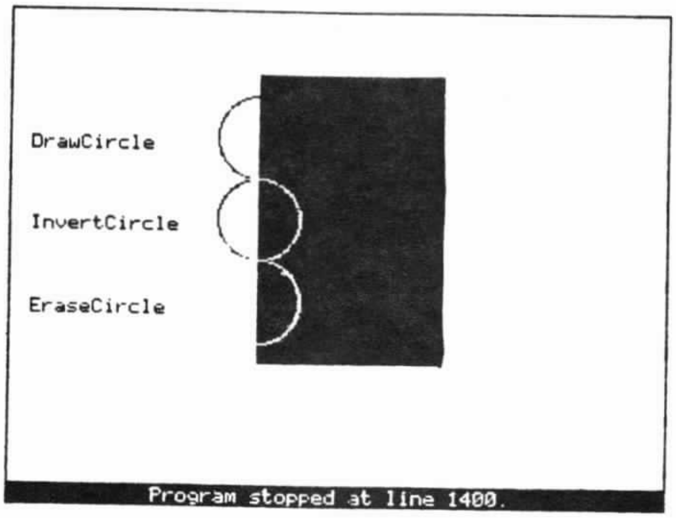
This chapter discusses GRiDBASIC's graphics statements. With these statements you can draw, invert, erase, and four figures:

- The box
- The circle
- The dot
- The line

This manual has one example each for the circle, dot, and line statements (for DRAW, INVERT, and ERASE). Each example shows how the graphic appears after a particular statement by placing part of the graphic against the screen and part against a white box. See Figures 10-1, 10-2, and 10-3 below. We place these examples at the front of the chapter for easy access and comparison. The programs that generated these figures are listed later in this with each relevant statement.

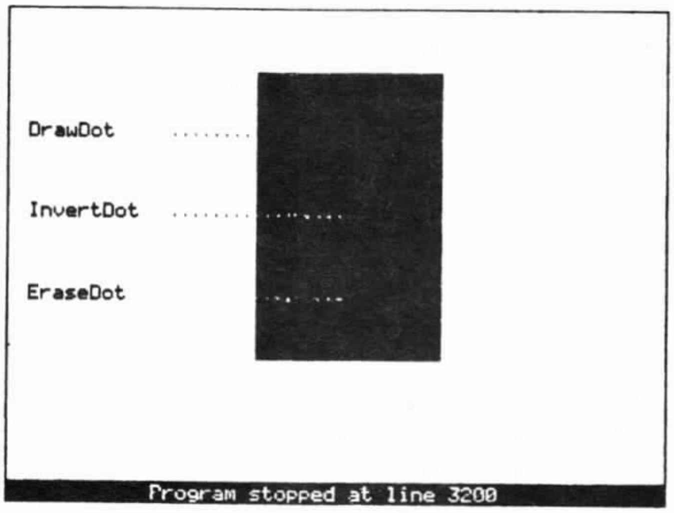
You can also position character strings (DrawChars), create menus, move boxes, and place prompt messages.

In graphics syntax, x and y represent (respectively) the horizontal and vertical coordinates of the point being described. These points are screen bits or "pixels." The screen is 320 pixels wide and 240 pixels deep.



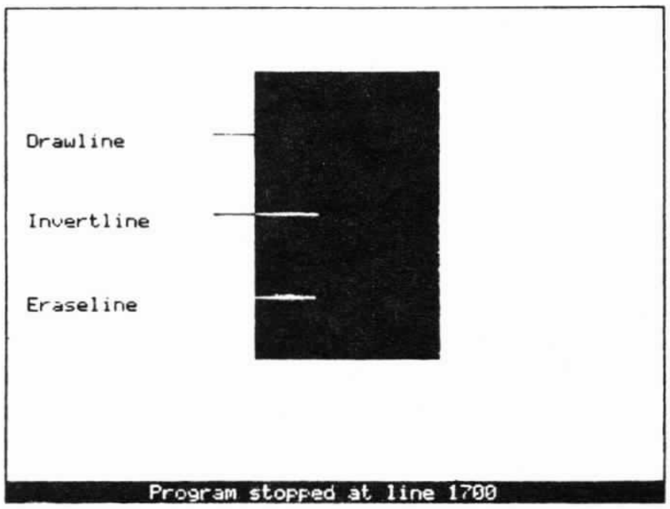
```
Program stopped at line 1400.
```

Figure 10-1. The Three Circle Graphics



```
Program stopped at line 3200
```

Figure 10-2. The Three Dot Graphics



```
Program stopped at line 1700
```

Figure 10-3. The Three Line Graphics

CLEARMSG

The clear message statement.

FORMAT

CLEARMSG

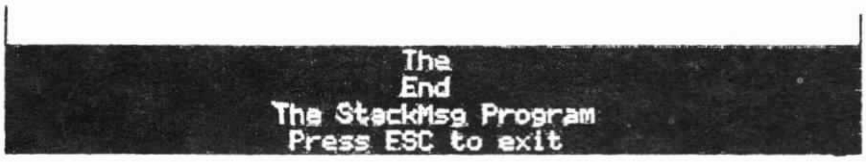
NOTES

CLEARMSG clears any prompt previously specified by the STACKMSG statement. If you include a STACKMSG prompt inside a loop or want to move on to a new message, you must clear the old message with CLEARMSG. If you don't, messages stack up, as shown in Figure 10-4. See STACKMSG later in this chapter for details.

EXAMPLE

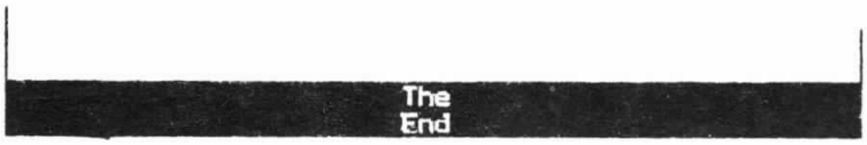
```
1000 PRINT "Press a key"
1100 STACKMSG "Press ESC to exit"
1200 STACKMSG "The StackMsg Program"
1300 LET Key$=INKEY$
1400 IF Key$="" THEN GOTO 1300 ELSE PRINT "Key is: "; Key$
1500 CLEARMSG
1600 STACKMSG "End
1700 STACKMSG "The
```

If it weren't for line 1500, the prompt would look like the one in Figure 10-4. CLEARMSG clears the first prompt so that the final prompt looks like the one in Figure 10-5.



```
The
End
The StackMsg Program
Press ESC to exit
```

Figure 10-4. Before CLEARMSG



```
The
End
```

Figure 10-5. After CLEARMSG

DOMENU

This statement creates a menu.

FORMAT

```
variable=DOMENU(prompt$, choice$|choice$[|choice$] ...)
```

NOTES

The DOMENU statement draws a menu at the bottom of the screen. This menu resembles the ones you have seen in GRiD applications. DOMENU asks you to specify the prompt message (prompt$) at the bottom of the screen and the various choices the menu will offer.

Separate each choice with a bar (|) by pressing CODE-SHIFT-; . DOMENU assigns a number to each choice, the first choice is 1, the second 2, etc. In this way, you can execute the choice with an ON GOTO or ON GOSUB statement. Only the size of the screen limits the number of choices you can present.

EXAMPLE

```
1000 LET Picky$="Make your play"
1100 LET Yours=DOMENU(Picky$,"Stand and fight|Flee and retreat|Buy
'em out")
1200 IF Yours=0 THEN LET Picky$="Oh no you don't: Choose"
1300 ON Yours GOTO 1500,1600,1700
1400 END
1500 PRINT "Fire when ready, Gridley":END
1600 PRINT "Come back, come back, come....":END
1700 PRINT "Okay. Let's talk, turkey":END
```

This example presents a menu as shown in Figure 10-6. "Picky$" is string variable to which we assign the prompt ("Make your play").

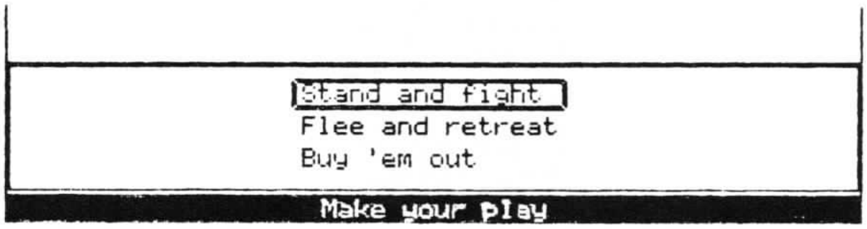


Figure 10-6. A Menu Created with DOMENU

DRAWBOX

The DRAWBOX statement draws a solid (light-colored) rectangle.

FORMAT

```
DRAWBOX topLeft (x,y) extent (x,y)
```

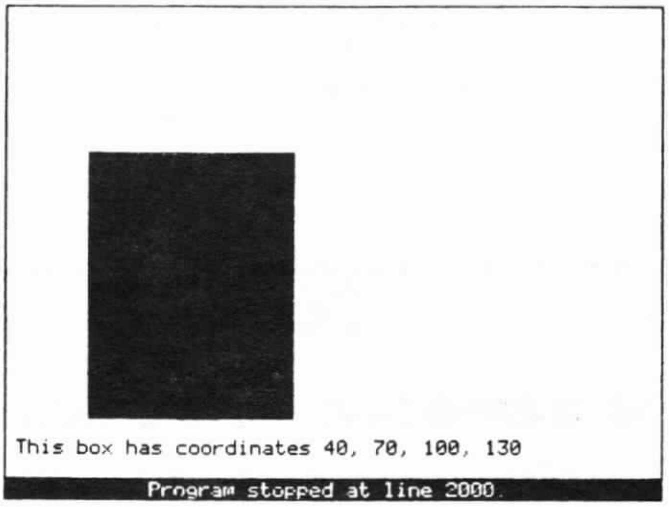
NOTES

DRAWBOX needs four coordinates. The first two describe the top left corner of the box. The second two describe the horizontal and vertical extensions from the starting point.

EXAMPLE

```
1000 INPUT "Top left horizontal coordinate";A
1100 INPUT "Top left vertical coordinate";B
1200 INPUT "Extend how far horizontally";C
1300 INPUT "Extend how far vertically";D
1400 EraseBox 0,0,320,240
1500 DrawBox A, B, C, D
1600 FOR Pause=1 TO 100: NEXT Pause
1700 LOCATE 5,210
1800 PRINT "This box has coordinates "; A;", ";B;", ";C;", ";D
1900 FOR Pause=1 TO 100: NEXT Pause
2000 END
```

This program asks you to describe a box and then draws that box. See Figure 10-7 for an example.



```
This box has coordinates 40, 70, 100, 130
Program stopped at line 2000.
```

Figure 10-7. An Example of DRAWBOX

DRAWCHARS

DRAWCHARS places characters on the screen at the stated coordinates.

FORMAT

```
DRAWCHARS string x,y
```

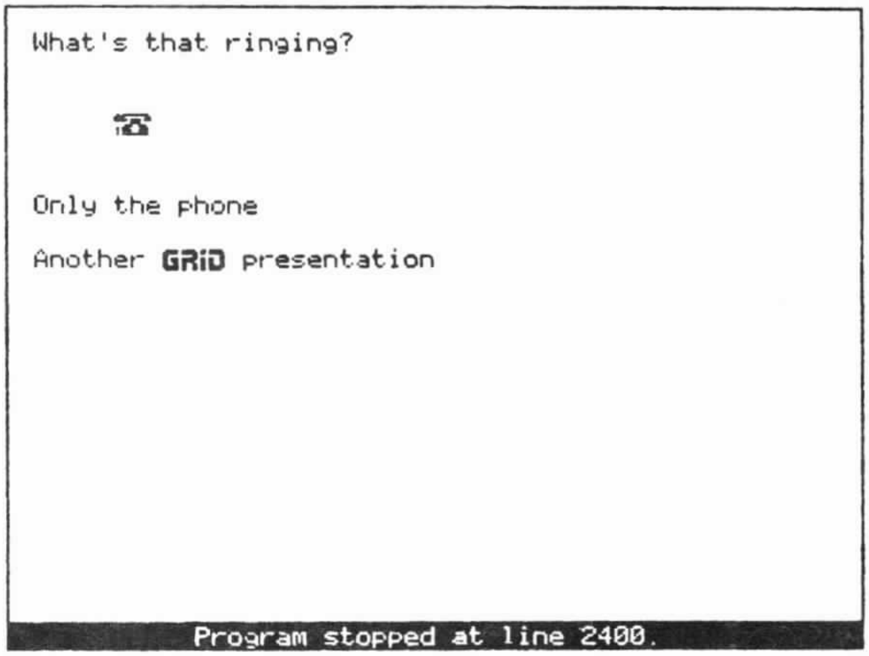
NOTES

The coordinates in DRAWCHARS define the upper left pixel of the first character in the string. This statement accepts strings surrounded by quotation marks, strings defined by the CHR$ statement and ASCII numbers, or a combination of the two. You cannot join strings with the semicolon (as with the PRINT statement. Instead, you must always concatenate them with the plus sign (+).

EXAMPLE

```
1000 DRAWCHARS "What's that ringing?",10,10
1100 FOR Starts=1 TO 100: NEXT Starts
1200 FOR Phone=1 TO 3
1300 LET Ring=1
1400 WHILE Ring<30
1500 DRAWCHARS CHR$(142)+CHR$(143),40,40
1600 DRAWCHARS CHR$(142)+CHR$(143),42,40
1700 LET Ring=Ring+1
1800 WEND
1900 FOR Time=1 TO 100: NEXT Time
2000 NEXT Phone
2100 DRAWCHARS "Only the phone",10,70
2200 FOR Pause=1 TO 100: NEXT Pause
2300 DRAWCHARS "Another "+ CHR$(137)+CHR$(138)+CHR$(139)+CHR$(140)+
" presentation",10, 90
2400 END
```

Lines 1000 and 2100 demonstrate placement of a string enclosed in quotation marks. Lines 1500 and 1600 show concatenation of individual ASCII characters (Line 1600 the string repositions the string for an animation effect). Line 2300 combines both quotes and ASCII codes to print the program's final message. The loops at lines 1100 and 2200 delay execution of the program for another effect. See Figure 10-8 for a picture of this program.



```
What's that ringing?

     ☎

Only the phone

Another GRiD presentation
```

```
Program stopped at line 2400.
```

Figure 10-8. The DRAWCHARS Example

DRAWCIRCLE

This statement positions and draws the outline of a circle.

FORMAT

DRAWCIRCLE x,y, radius

NOTES

The x,y coordinates specify the center of the circle. The radius is measured in screen bits.

EXAMPLE

```
1000 DRAWBOX 120, 30, 90, 140
1100 DRAWCIRCLE 120,60,20
1200 LOCATE 10, 60: PRINT "DrawCircle"
1300 INVERTCIRCLE 120,100,20
1400 LOCATE 10, 100: PRINT "InvertCircle"
1500 ERASECIRCLE 120, 140, 20
1600 LOCATE 10,140: PRINT "EraseCircle"
1700 END
```

DRAWDOT

The DRAWDOT statement turns on one screen bit (also known as a "pixel").

FORMAT

```
DRAWDOT x,y
```

NOTES

The two arguments are the dot's horizontal and vertical coordinates.

EXAMPLE

```
1000 DRAWBOX 120, 30, 90, 140
1100 LOCATE 10, 55: PRINT "DrawDot"
1200 LOCATE 10, 95: PRINT "InvertDot"
1300 LOCATE 10, 135: PRINT "EraseDot"
1400 REM The DrawDot Routine
1500 LET X=80: Y=60
1600 WHILE X<=160
1700 DRAWDOT X,Y
1800 LET X=X+5
1900 WEND
2000 REM The InvertDot Routine
2100 LET X=80: LET Y=100
2200 WHILE X<=160
2300 INVERTDOT X,Y
2400 LET X=X+5
2500 WEND
2600 REM The EraseDot Routine
2700 LET X=80:Y=140
2800 WHILE X<=160
2900 ERASEDOT X,Y
3000 LET X=X+5
3100 WEND
3200 END
```

This program differs from programs for the other figures in order to put five pixels between the dots. Without these spaces, you cannot tell the difference between similar statements for DOT and LINE.

DRAWLINE

The DRAWLINE statement draws a line.

FORMAT

DRAWLINE startPoint (x,y) endpoint (x,y)

NOTES

DRAWLINE needs four arguments -- the horizontal and vertical points for the start of the line and the horizontal and vertical points for the end of the line.

EXAMPLE

```
1000 DRAWBOX 120, 30, 90, 140
1100 DRAWLINE 100, 60, 150, 60
1200 LOCATE 10, 60: PRINT "Drawline"
1300 INVERTLINE 100, 100, 150, 100
1400 LOCATE 10, 100: PRINT "Invertline"
1500 ERASELINE 100, 140, 150, 140
1600 LOCATE 10,140: PRINT "Eraseline"
1700 END
```

ERASEBOX

This statement erases a box in the position described by its coordinates.

FORMAT

ERASEBOX topLeft (x,y) extent (x,y)

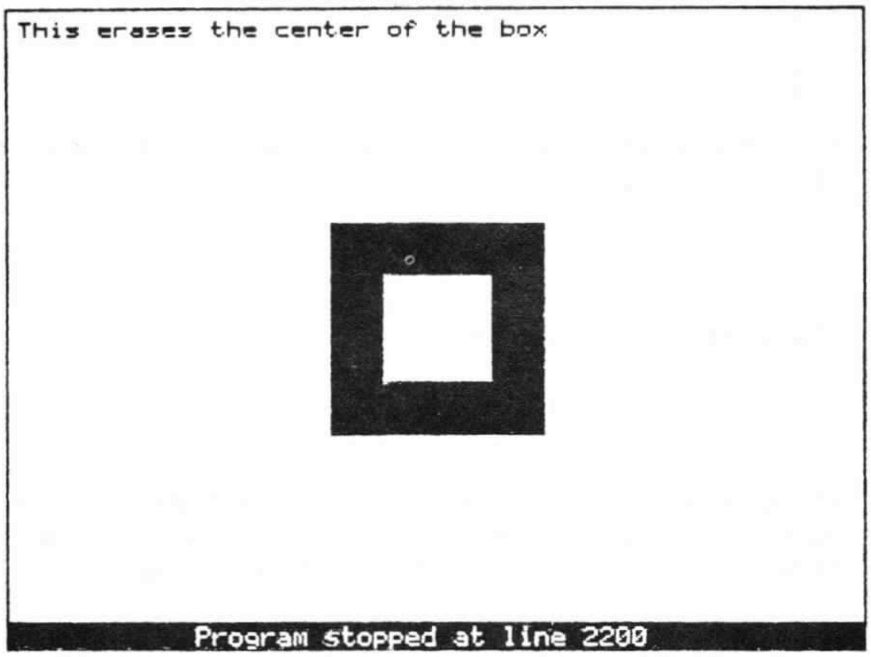
NOTES

You cannot see ERASEBOX working against a dark background. Only against a light background. The act of erasing only turns screen bits off.

EXAMPLE

```
1000 LET A=120: B=80: C=80: D=80
1100 DrawBox A,B,C,D
1200 FOR Pause=1 TO 200: NEXT Pause
1300 LET A=140: B=100: C=40: D=40
1400 PRINT "This erases the center of the box"
1500 GOSUB 2000
1600 PRINT "And this erases everything."
1700 LET A=0: B=0: C=320: D=240
1800 GOSUB 2000
1900 END
2000 FOR Pause=1 TO 200: NEXT Pause
2100 EraseBox A,B,C,D
2200 FOR Pause=1 TO 200: NEXT Pause
2300 RETURN
```

This example draws a box, erases its center, and then clears (erases) the entire screen. Figure 10-9 shows the program run through the first erasure.



```
This erases the center of the box
```

```
Program stopped at line 2200
```

Figure 10-9. An Example of ERASEBOX

ERASECIRCLE

This statement erases a circle of the size and position described by its coordinates.

FORMAT

```
ERASECIRCLE x,y, radius
```

NOTES

You cannot see the circle described by ERASECIRCLE unless you erase over a white area. The x,y coordinates specify the center of the circle. The radius is measured in screen bits.

EXAMPLE

```
1000 DRAWBOX 120, 30, 90, 140
1100 DRAWCIRCLE 120,60,20
1200 LOCATE 10, 60: PRINT "DrawCircle"
1300 INVERTCIRCLE 120,100,20
1400 LOCATE 10, 100: PRINT "InvertCircle"
1500 ERASECIRCLE 120, 140, 20
1600 LOCATE 10,140: PRINT "EraseCircle"
1700 END
```

ERASEDOT

This statement erases a dot.

FORMAT

```
ERASEDOT x,y
```

NOTES

ERASEDOT turns off one screen bit. It is only visible when the dot resides on a light background.

EXAMPLE

```
1000 DRAWBOX 120, 30, 90, 140
1100 LOCATE 10, 55: PRINT "DrawDot"
1200 LOCATE 10, 95: PRINT "InvertDot"
1300 LOCATE 10, 135: PRINT "EraseDot"
1400 REM The DrawDot Routine
1500 LET X=80: Y=60
1600 WHILE X<=160
1700 DRAWDOT X,Y
1800 LET X=X+5
1900 WEND
2000 REM The InvertDot Routine
2100 LET X=80: LET Y=100
2200 WHILE X<=160
2300 INVERTDOT X,Y
2400 LET X=X+5
2500 WEND
2600 REM The EraseDot Routine
2700 LET X=80:Y=140
2800 WHILE X<=160
2900 ERASEDOT X,Y
3000 LET X=X+5
3100 WEND
3200 END
```

This program differs from programs for the other figures in order to put five pixels between the dots. Without these spaces, you cannot tell the difference between similar statements for DOT and LINE.

ERASELINE

This statement erases a line in the position described by its coordinates.

FORMAT

```
ERASELINE startPoint (x,y) endPoint (x,y)
```

NOTES

ERASELINE needs four arguments -- the horizontal and vertical points for the start of the erasure and the horizontal and vertical points for ending erasure.

EXAMPLE

```
1000 DRAWBOX 120, 30, 90, 140
1100 DRAWLINE 100, 60, 150, 60
1200 LOCATE 10, 60: PRINT "Drawline"
1300 INVERTLINE 100, 100, 150, 100
1400 LOCATE 10, 100: PRINT "Invertline"
1500 ERASELINE 100, 140, 150, 140
1600 LOCATE 10,140: PRINT "Eraseline"
1700 END
```

INVERTBOX

This statement inverts the "colors" of a box in the position described by its coordinates.

FORMAT

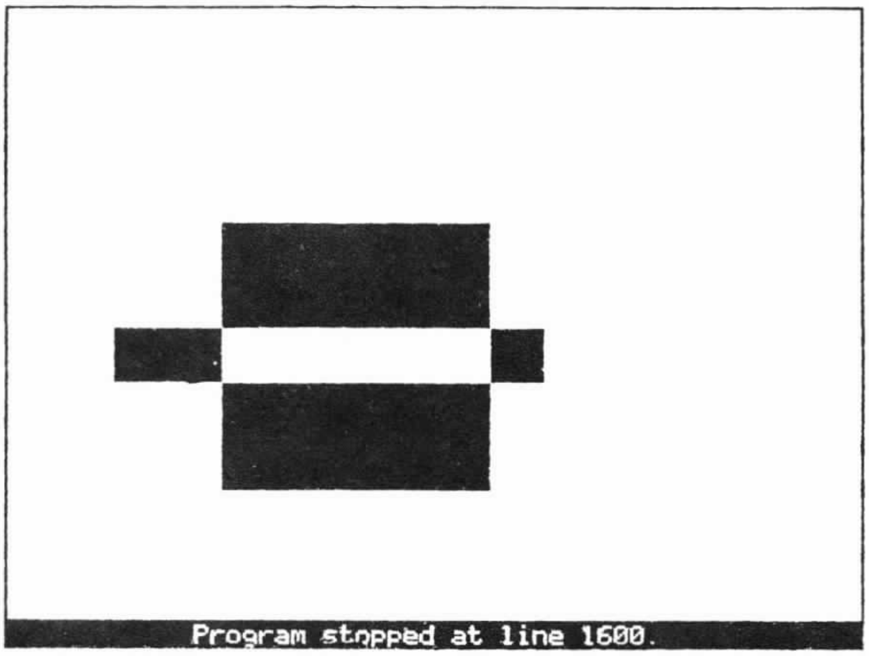
```
INVERTBOX TopLeft (x,y) Extent (x,y)
```

NOTES

The inversion here amounts to turning off bits that are on and turning on bits that are off. As a result, INVERTBOX describes a light rectangle on a dark background and a dark rectangle against a light background. See Figure 10-10 below.

EXAMPLE

```
1000 DRAWBOX 80,80,100,100
1100 LET A=40: B=100: C=1: D=1
1200 WHILE A<200
1300 INVERTBOX A,120,5,20
1400 A=A+5
1500 WEND
1600 END
```



```
Program stopped at line 1600.
```

Figure 10-10. An Example of INVERTBOX

INVERTCIRCLE

This statement positions and draws a circle the "colors" of which are the opposite of the background area.

FORMAT

```
INVERTCIRCLE x,y, radius
```

NOTES

The inversion here amounts to turning off bits that are on and turning on bits that are off. As a result, INVERTCIRCLE describes a light circle on a dark background and a dark circle against a light background. The x,y coordinates specify the center of the circle. The radius is measured in screen bits.

EXAMPLE

```
1000 DRAWBOX 120, 30, 90, 140
1100 DRAWCIRCLE 120,60,20
1200 LOCATE 10, 60: PRINT "DrawCircle"
1300 INVERTCIRCLE 120,100,20
1400 LOCATE 10, 100: PRINT "InvertCircle"
1500 ERASECIRCLE 120, 140, 20
1600 LOCATE 10,140: PRINT "EraseCircle"
1700 END
```

INVERTDOT

INVERTDOT draws a dot of a "color" opposite that of its background.

FORMAT

INVERTDOT topLeft (x,y) extent (x,y)

NOTES

INVERTDOT places a light dot on a dark background and a dark dot on a light background. If a screen bit is on, INVERTDOT turns it off. If off, it turns it on.

EXAMPLE

```
1000 DRAWBOX 120, 30, 90, 140
1100 LOCATE 10, 55: PRINT "DrawDot"
1200 LOCATE 10, 95: PRINT "InvertDot"
1300 LOCATE 10, 135: PRINT "EraseDot"
1400 REM The DrawDot Routine
1500 LET X=80: Y=60
1600 WHILE X<=160
1700 DRAWDOT X,Y
1800 LET X=X+5
1900 WEND
2000 REM The InvertDot Routine
2100 LET X=80: LET Y=100
2200 WHILE X<=160
2300 INVERTDOT X,Y
2400 LET X=X+5
2500 WEND
2600 REM The EraseDot Routine
2700 LET X=80:Y=140
2800 WHILE X<=160
2900 ERASEDOT X,Y
3000 LET X=X+5
3100 WEND
3200 END
```

This program differs from programs for the other figures in order to put five pixels between the dots. Without these spaces, you cannot tell the difference between similar statements for DOT and LINE.

INVERTLINE

INVERTLINE draws a line of a "color" opposite that of its background.

FORMAT

```
INVERTLINE startPoint (x,y) endPoint (x,y)
```

NOTES

INVERTLINE needs four arguments -- the horizontal and vertical points for the start of the line and the horizontal and vertical points for the end of the line.

The inversion here amounts to turning off bits that are on and turning on bits that are off. As a result, INVERTLINE describes a light line on a dark background and a dark line against a light background.

EXAMPLE

```
1000 DRAWBOX 120, 30, 90, 140
1100 DRAWLINE 100, 60, 150, 60
1200 LOCATE 10, 60: PRINT "Drawline"
1300 INVERTLINE 100, 100, 150, 100
1400 LOCATE 10, 100: PRINT "Invertline"
1500 ERASELINE 100, 140, 150, 140
1600 LOCATE 10,140: PRINT "Eraseline"
1700 END
```

MOVEBOX

This statement copies an existing box to a second set of coordinates.

FORMAT

```
MOVEBOX topLeft (x,y) extent (x,y) destination topLeft (x,y)
```

NOTES

MOVEBOX copies, but does not erase, an existing box. If you need to give the illusion of movement you must follow your MOVEBOX statement with an ERASEBOX statement (the erase coordinates should be those of the original box).

EXAMPLE

```
1000 LET A=100: B=80: C=80: D=80
1100 DrawBox A,B,C,D
1200 FOR Pause=1 TO 200: NEXT Pause
1300 MoveBox A,B,C,D,200,80
1400 EraseBox A,B,C,D
1500 END
```

STACKMSG

This statement places a message in inverse video at the bottom of the screen.

FORMAT

```
STACKMSG "PromptString"
[STACKMSG "PromptString"]
```

NOTES

STACKMSG takes only one parameter -- the string character that constitutes the prompt message. If you have a second STACKMSG statement, the message area will expand to hold both messages. NOTE: When writing two messages, place the first message second. For example, the messages in

```
1000 STACKMSG "Press ESC to exit"
1100 STACKMSG "The StackMsg Program"
```

appear in reverse vertical order:

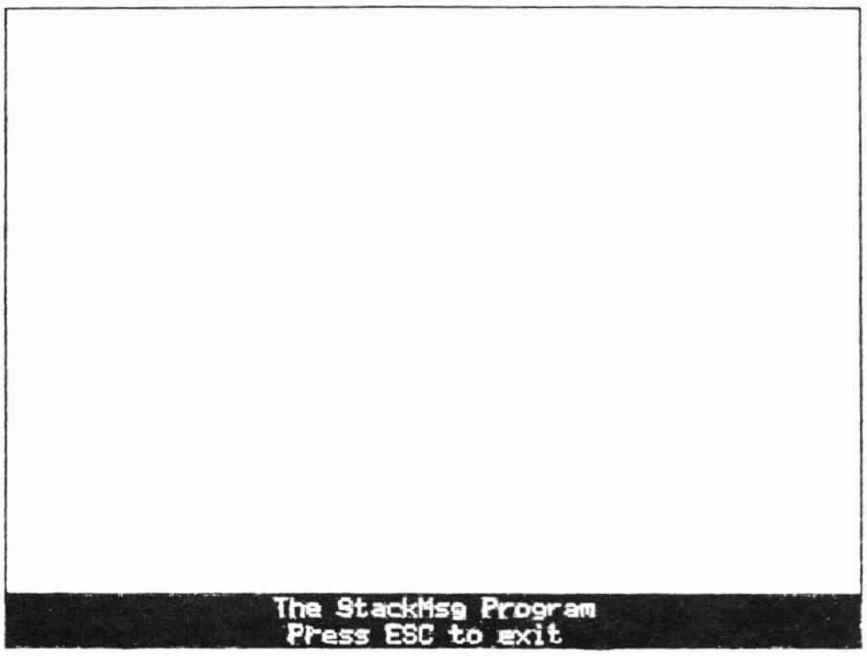
```
The StackMsg Program
 Press ESC to exit
```

STACKMSG messages remain on the screen for a split second. To make them stay longer, you can follow the STACKMSG statement with some kind of loop. The most common loop displays the prompt until someone presses a key. The message then disappears as program execution continues. NOTE: Do not include a STACKMSG prompt inside a loop unless you follow it immediately with a CLEARMSG statement. Otherwise, the prompt area scrolls up the screen. Adding a third and/or fourth message without CLEARMSG also causes scrolling. See CLEARMSG earlier in this chapter for details.

EXAMPLE

```
1000 STACKMSG "Press ESC to exit"
1100 STACKMSG "The StackMsg Program"
1200 LET Key$=INKEY$
1300 IF Key$ <> "" THEN PRINT Key$; " gets me out of the loop" ELSE
GOTO 1200
1400 END
```

In this example, lines 1000 and 1100 set up a two-line prompt. Lines 1200 and 1300 create an INKEY$ loop that waits for a key press to occur. When someone presses a key, line 1300 prints the key's character.



The StackMsg Program
Press ESC to exit

Figure 10-11. A STACKMSG Prompt Line

APPENDIX A: ERROR MESSAGES

GRiDBASIC error messages are listed here in alphabetical order.

Array is too large

What happened	You tried to put over 65,535 bytes into an array.
What to do	Redimension the array so that its size falls within legal limits.

Array reference is out of range

What happened	You probably have a subscript of 0; you dimensioned an array with a variable and still haven't assigned a number to that variable. Or, you have assigned a number greater than the subscript allows.
What to do	Check your subscripts, especially those that are variables. Remember: You can only have ten items in an array without dimensioning.

Attempt to read past end of file

What happened	An INPUT# statement is executed after all the data in a file has already been input, or the file is a null (empty) file.
What to do	Place the EOF function in your program to detect end

of file and avoid this error.

ELSE encountered without matching IF

What happened	You programmed an IF THEN ELSE, but managed to leave out the IF.
What to do	Put in the IF statement.

Empty line

What happened	This is a system-level error.
What to do	Nothing. You won't see this error.

Expression error

What happened	This is a system-level error.
What to do	Nothing. You won't see this error.

File already open

What happened	This run-time error occurs when you try to reopen a file you've already opened.
What to do	Check to see what file is open and its tag number. Also, is it a random access file? Sometimes you try to open a different file, but give a tag that is already in use. You can either write a CLOSE statement for the file or (if the tag is the problem) give the correct number. If the file is random access, make sure you haven't tried to read or write to it as you would with a sequential file.

File is not open for random access.

What happened	This run-time error indicates you've tried to read from or write to a file (with GET or PUT), but you haven't opened it as a random access file.
What to do	Check to see if you've opened the file. If you have, check the file tag number and the number you've used with the your access command. Also, did you assign the file the "R" (for "random") mode? And remember:

Random access files require a FIELD statement to allot buffer space.

File not open

What happened	This run-time error indicates you've tried to read from or write to a file you haven't opened.
What to do	Check to see if you've opened the file. If you have, check the file tag number and the number you've used with the your access command.

FOR encountered without matching NEXT

What happened	You began a FOR NEXT loop, but failed to complete it with a NEXT statement.
What to do	Locate the place where the loop should end and put in the NEXT statement with appropriate variable. Or erase the FOR TO [STEP] if you no longer want the loop in your program.

Generation error

What happened	This is a system-level error.
What to do	Nothing. You won't see this error.

Illegal character

What happened	This error message is reserved for later use.
What to do	Nothing. It won't happen.

Illegal value

What happened	A number that is either too large or too small causes this at the system level.
What to do	Nothing. You won't see this error.

Improper expression

What happened	This is a system-level error.
What to do	Nothing. You won't see this error.

Improper function call

What happened	This message covers a multitude of sins -- from usin non-existent or unimplemented functions.
What to do	Check to see that your program contains only current functions. Are their names correctly spelled? If everything looks okay, try re-running the program. If that fails, reboot the system and then re-run the program.

Improper loop nesting

What happened	You written an inner loop and an outer loop so that they overlap. A run-time error.
What to do	Untangle the offending loops so that no overlapping takes place. See FOR NEXT and WHILE WEND for details.

Improper parameter in function call

What happened	This run-time error usually indicates an improper number of parameters or parentheses.
What to do	Check to see that you gave the correct number of paramenters and parentheses to the function.

Improper syntax

What happened	This is the catch-all phrase for any syntax problem. It occurs while programming, when you press RETURN, CODE-RETURN, or either vertical arrow key.
What to do	Check the syntax of all statements and functions on the current line and confirm the line again to see if the error remains.

Invalid variable

What happened	The interpreter has encountered either a variable with an illegal character in it (see Chapter Two) or the name of a file the system can't find.
What to do	Check the variables on the current line. If you are making a file reference, make sure that it is on the

device, under the subject, and of the kind you have named.

Mismatched quotes

What happened	This error occurs while you're programming and indicates you don't have the proper number of quotes on the current line.
What to do	Check quotation marks (") to make sure you have the correct number. NOTE: Don't try to put double quotes within double quotes. You can, however, put single quotes (') within double quotes.

Missing parameter in array reference

What happened	During programming, you have omitted one of the dimensions that you declared when dimensioning (DIM) the array.
What to do	Find the erring array and insert the missing parameter.

Missing parameter in function reference

What happened	While programming, you have omitted a required parameter from a function.
What to do	Find the function and determine which parameter is missing. Then insert the parameter.

NEXT encountered without matching FOR

What happened	You have failed to include the upper portion of the FOR NEXT loop -- FOR TO [STEP]. Or you have given the wrong variable after NEXT.
What to do	Put the FOR TO [STEP] portion of the loop at its proper place in your program. Or, if the wrong variable follows NEXT, correct it. Or, if the NEXT is an unwanted leftover, erase it.

Number of array dimensions disagrees with definition

What happened	You have given an array the wrong number of dimensions. This message can occur while programming or at run-time.

What to do	Check any array(s) in the current line and determine which has an improper number of dimensions.

Not implemented

What happened	You have used a word that GRiDBASIC has reserved for later use, but which does not yet work as a statement, function, or constant.
What to do	Figure out some other way to accomplish the purpose achieved by the unimplemented word.

Number of parameters disagrees with definition

What happened	You have given an array or a function the wrong number of parameters. This message can occur while programming or at run-time.
What to do	Check any array(s) or functions in the current line and determine which has an improper number of parameters.

Out of memory

What happened	You dimensioned an array so that it takes more memory than the system offers.
What to do	Redimension the offending array.

Ran out of data

What happened	A READ statement read all the available DATA items.
What to do	Add more data. Or put in a counter that causes the program to stop reading before exhausting the items in the data statement. Or put in a RESTORE statement to cause the data to be reread.

RETURN encountered outside subroutine

What happened	You have a RETURN statement that lacks a preceding matching GOSUB statement.
What to do	Either write the appropriate GOSUB statement or erase the RETURN.

Statement with syntax errors encountered

What happened	This message repeats at run-time what you saw as "Improper syntax" while programming. This means that you didn't correct the error.
What to do	Check the syntax of all statements and functions on the current line and confirm the line again to see if the error remains.

Type mismatch

What happened	Every variable and most operations expect a particular "type" of data -- string, numeric, or Boolean. Giving a foreign datum to a variable or operation causes this error. For example, giving a string to a numeric operator or variable.
What to do	Find the offending datum (or its source). Then change either the datum or the receiving statment so that a type match occurs.

Undefined line number

What happened	You have placed a line number in a statement (such as GOTO or GOSUB) for which no matching line exists.
What to do	Change the line number to point to the proper line. Or erase the pointer statement.

Variable expected here

What happened	You type a statement requiring a varialbe (such as INPUT) while programming, but didn't include its variable.
What to do	Find the statement and enter the variable(s).

WEND encountered without matching WHILE

What happened	A WHILE WEND loop lacks its WHILE statement.
What to do	Either insert the WHILE statement with its condition or erase WEND statement.

WHILE encountered without watching WEND

What happened	A WHILE WEND loop lacks its WEND statement.
What to do	Either insert the WEND statement with its condition or erase WHILE statement.

APPENDIX B: ASCII CHARACTERS

This appendix contains the ASCII (American Standard Code for Information Interchange) character codes. Programmers use these codes in everything from string handling functions (see Chapter Six) to communications work.

DEC	HEX	GRPH	ABBR	NAME	PRESS
00	00	␀	NUL	null	CTRL-SHIFT-2
01	01	␁	SOH	start of heading	CTRL-A
02	02	␂	STX	start of text	CTRL-B
03	03	␃	ETX	end of text	CTRL-C
04	04	␄	EOT	end of transmission	CTRL-D
05	05	␅	ENQ	enquiry	CTRL-E
06	06	␆	ACK	acknowledge	CTRL-F
07	07	␇	BEL	bell	CTRL-G
08	08	␈	BS	backspace	CTRL-H
09	09	▶	HT	horizontal tab	CTRL-I, TAB
10	0A	␊	LF	linefeed	CTRL-J
11	0B	␋	VT	vertical tab	CTRL-K
12	0C	␌	FF	form feed	CTRL-L
13	0D	␍	CR	carriage return	CTRL-M
14	0E	␎	SO	shift out	CTRL-N
15	0F	␏	SI	shift in	CTRL-O
16	10	␐	DLE	data link escape	CTRL-P
17	11	␑	DC1	device control 1(XON)	CTRL-Q
18	12	␒	DC2	device control 2	CTRL-R
19	13	␓	DC3	device control 3(XOFF)	CTRL-S
20	14	␔	DC4	device control 4	CTRL-T
21	15	␕	NAK	negative ack	CTRL-U
22	16	␖	SYN	synchronous idle	CTRL-V
23	17	␗	ETB	end trans. block	CTRL-W
24	18	␘	CAN	cancel	CTRL-X
25	19	␙	EM	end medium	CTRL-Y
26	1A	␚	SUB	substitute	CTRL-Z
27	1B	␛	ESC	escape	CTRL-;
28	1C	␜	FS	file separator	CTRL-SHIFT-,
29	1D	␝	GS	group separator	CTRL-=
30	1E	␞	RS	record separator	CTRL-SHIFT-.
31	1F	␟	US	unit separator	CTRL-SHIFT-hyphen
32	20		SP	space	
33	21	!		exclamation	
34	22	"		quotation marks	
35	23	#		number sign	
36	24	$		dollar sign	
37	25	%		percent sign	
38	26	&		ampersand	
39	27	'		apostrophe	
40	28	(		opening parenthesis	
41	29	)		closing parenthesis	
42	2A	*		asterisk	
43	2B	+		plus	
44	2C	,		comma	
45	2D	-		hyphen	
46	2E	.		period	
47	2F	/		slash	
48	30	0			
49	31	1			
50	32	2			
51	33	3			
52	34	4			
53	35	5			
54	36	6			
55	37	7			
56	38	8			
57	39	9			
58	3A	:		colon	
59	3B	;		semicolon	
60	3C	<		less than	
61	3D	=		equal to	
62	3E	>		greater than	
63	3F	?			
64	40	@		commerical at sign	

DEC	HEX	GRPH	ABBR	NAME	PRESS
65	41	A			
66	42	B			
67	43	C			
68	44	D			
69	45	E			
70	46	F			
71	47	G			
72	48	H			
73	49	I			
74	4A	J			
75	4B	K			
76	4C	L			
77	4D	M			
78	4E	N			
79	4F	O			
80	50	P			
81	51	Q			
82	52	R			
83	53	S			
84	54	T			
85	55	U			
86	56	V			
87	57	W			
88	58	X			
89	59	Y			
90	5A	Z			
91	5B	[		opening bracket	CODE-,
92	5C	\		backslash	CODE-SHIFT-,
93	5D	]		closing bracket	CODE-
94	5E	^		circumflex	
95	5F	_		underline	
96	60	`		back quote	CODE-'
97	61	a			
98	62	b			
99	63	c			
100	64	d			
101	65	e			
102	66	f			
103	67	g			
104	68	h			
105	69	i			
106	6A	j			
107	6B	k			
108	6C	l			
109	6D	m			
110	6E	n			
111	6F	o			
112	70	p			
113	71	q			
114	72	r			
115	73	s			
116	74	t			
117	75	u			
118	76	v			
119	77	w			
120	78	x			
121	79	y			
122	7A	z			
123	7B	{		left curly bracket	CODE-SHIFT-,
124	7C	|		vertical line	CODE-SHIFT-;
125	7D	}		right curly bracket	CODE-SHIFT-.
126	7E	~		tilde	CODE-;
127	7F	■	DEL	delete	CODE-SHIFT-hyphen

INDEX