INSERT

Chapter Eight

Special Features

OVERVIEW

BASIC-80 provides the programmer with several special features. One of these features, Error Trapping, is useful for detecting errors during program execution. Another feature is the PRINT USING statement. This statement allows the programmer to specify the format of both numeric and string output.

Another important feature is the Trace flag, which allows the programmer to follow, line-by-line, the execution of a program.

BASIC-80 also provides the facilities for overlay management. The CHAIN and COMMON statement are used for this function.

ERROR TRAPPING

BASIC-80 allows the programmer to write error detection and error handling routines which can attempt to recover from errors, or provide more complete explanations of the causes of errors. This facility has been added through the use of the ON ERROR GOTO, RESUME, and ERROR statements, and with the ERR and ERL variables.

ON ERROR GOTO (enable error trapping)

Form: ON ERROR GOTO <line number>

The ON ERROR GOTO statement is used to enable error trapping and specify the first line of the error handling subroutine.

Once error trapping has been enabled, all errors detected, including Command Mode errors (e.g., Syntax errors), will cause a jump to the specified error handling subroutine. If <line number> does not exist, an "Undefined line number" error results.

To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes BASIC-80 to stop and print the error message for the error that caused the trap. We recommend that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

If an error occurs during execution of an error handling subroutine, the BASIC error message is printed and execution terminates. Error trapping does not trap errors within the error handling subroutine.

Example:

```
10 ON ERROR GOTO 1000
```

RESUME (continue execution)

Forms:

RESUME
RESUME 0
RESUME NEXT
RESUME <line number>

The RESUME statement is used to continue program execution after an error recovery procedure has been performed.

Any one of the four formats shown above may be used, depending upon where execution is to resume:

RESUME or RESUME 0	Execution resumes at the statement which caused the error.
RESUME NEXT	Execution resumes at the statement immediately following the one which caused the error.
RESUME<line number>	Execution resumes at <line number>.

A RESUME statement that is not in an error trap routine causes a "RESUME without error" message to be printed.

Error Trap Example:

```
100 ON ERROR GOTO 500
200 INPUT"WHAT ARE THE NUMBERS TO DIVIDE";X,Y
210 Z=X/Y
220 PRINT "QUOTIENT IS";Z
230 GOTO 200
500 IF ERR=11 AND ERL=210 THEN 520
510 STOP
520 PRINT"YOU CAN'T HAVE A DIVISOR OF ZERO!"
530 RESUME 200
```

ERROR (generate error)

Form: ERROR <integer expression>

The ERROR statement can be used either to simulate the occurrence of a BASIC-80 error, or to allow error codes to be defined by the user.

The value of <integer expression> must be greater than 0 and less than 255. If the value of <integer expression> equals an error code already in use by BASIC-80, the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be printed.

To define your own error code, use a value that is greater than any used by BASIC-80's error codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to BASIC-80.) This user-defined error code may then be conveniently handled in an error trap routine.

If an ERROR statement specifies a code for which no error message has been defined, BASIC-80 responds with the message "Unprintable error". Execution of an ERROR statement for which there is no error trap routine causes an error message to be printed and execution to halt.

Example:

```
LIST
10 S = 10
20 T = 5
30 ERROR S + T
40 END
Ok
RUN
String too long in line 30
```

Or, in Command Mode:

```
Ok
ERROR 15        (you type this line)
String too long (BASIC-80 types this line)
Ok
```

ERR and ERL Variables

When an error handling subroutine is entered, the variable ERR contains the error code for the error, and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF/THEN statements to direct program flow in the error trap routine.

If the statement that caused the error was a Command Mode statement, ERL will contain 65535. To test if an error occurred in a Command Mode statement, use IF 65535 = ERL THEN ... Otherwise, use

IF ERR = error code THEN ...

IF ERL = line number THEN ...

If the line number is not on the right side of the relational operator, it cannot be renumbered by RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement.

When the error handling subroutine is entered, the variable ERR contains the error code for the error. The error codes and their meanings are listed on the next page. See Appendix A, "Error Messages," for a more detailed discussion of the error messages.

ERROR CODES

General Errors

CODE	ERROR
1	NEXT WITHOUT FOR
2	SYNTAX ERROR
3	RETURN WITHOUT GOSUB
4	OUT OF DATA
5	ILLEGAL FUNCTION CALL
6	OVERFLOW
7	OUT OF MEMORY
8	UNDEFINED LINE
9	SUBSCRIPT OUT OF RANGE
10	DUPLICATE DEFINITION
11	DIVISION BY ZERO
12	ILLEGAL DIRECT
13	TYPE MISMATCH
14	OUT OF STRING SPACE
15	STRING TOO LONG
16	STRING FORMULA TOO COMPLEX
17	CAN'T CONTINUE
18	UNDEFINED USER FUNCTION
19	NO RESUME
20	RESUME WITHOUT ERROR
21	UNPRINTABLE ERROR
22	MISSING OPERAND
23	LINE BUFFER OVERFLOW
26	FOR WITHOUT NEXT
29	WHILE WITHOUT WEND
30	WEND WITHOUT WHILE

Table 8-1
Error Codes.

Disk Errors

CODE	ERROR
50	FIELD OVERFLOW
51	INTERNAL ERROR
52	BAD FILE NUMBER
53	FILE NOT FOUND
54	BAD FILE MODE
55	FILE ALREADY OPEN
57	DISK I/O ERROR
58	FILE ALREADY EXISTS
61	DISK FULL
62	INPUT PAST END
63	BAD RECORD NUMBER
64	BAD FILE NAME
66	DIRECT STATEMENT IN FILE
67	TOO MANY FILES

Table 8-1 (Cont'd.)
Error Codes.

FORMATTED OUTPUT

The PRINT USING statement can be used to output information in a specific format. This feature is useful in such applications as printing payroll checks or accounting reports.

PRINT USING (format output)

Form: PRINT USING<string exp>;<list of expressions>

The PRINT USING statement is used to print strings or numbers using a specified format.

<list of expressions> is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons. <string exp> is a string literal (or variable) that is comprised of special formatting characters. These formatting characters (see below) determine the field, and the format, of the printed strings or numbers.

String Fields

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

"!"

This specifies that only the first character in the given string is to be printed.

"\n spaces\"

This specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right.

Example:

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!";A$;B$
30 PRINT USING "\   \";A$;B$
40 PRINT USING "\     \;A$;B$;"!!"
RUN
LO
LOOKOUT
LOOK OUT !!
```

"&"

The ampersand specifies a variable length string field. When the field is specified with "&", the string is output exactly as input.

Example:

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!";A$
30 PRINT USING "&";B$
RUN
L
OUT
```

Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

"#"

A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

"."

A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded as necessary.

Examples:

```
PRINT USING "##.##";.78
 0.78

PRINT USING "###.##";987.654
987.65

PRINT USING "##.##   ";10.2,5.3,66.789,.234
10.20 5.30 66.79 0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

"+"

A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.

"–"

A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign. If the number is positive, a space will be printed.

```
PRINT USING "+##.##   ";-68.95,2.4,55.6,-.9
-68.95    +2.40     +55.60     -0.90

PRINT USING "##.##- ";-68.95,22.449,-7.01
68.95-     22.45       7.01-
```

"**"

A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

Example:

```
        PRINT USING "**#.#   ";12.39,-0.9,765.1
        *12.4    *-0.9    765.1
```

"$$"

A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

Example:

```
        PRINT USING "$$###.##";456.78
         $456.78
```

"**$"

The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

Example:

```
    PRINT USING "**$##.##";2.34
    ***$2.34
```

","

A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit on the left side of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (^^^^) format.

Example:

```
    PRINT USING "####,.##";1234.5
    1,234.50

    PRINT USING "####.##,";1234.5
    1234.50,
```

"^^^^"

Four carats (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carats allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

Example:

```
    PRINT USING "##.##^^^^";234.56
     2.35E+02

    PRINT USING ".####^^^^-";888888
     .8889E+06

    PRINT USING ""+.##^^^^";123
    +.12E+03
```

"_"

An underscore in the format string causes the next character to be output as a literal character.

Example:

```
PRINT USING "_!##.##_!";12.34
!12.34!
```

The literal character itself may be an underscore by placing "__" in the format string.

Errors

If the number to be printed is larger than the specified numeric field, a percent sign (%) is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

Example:

```
PRINT USING "##.##";111.22
%111.22

PRINT USING ".##";.999
%1.00
```

If the number of digits specified exceeds 24, an "Illegal function call" error will result.

TRACE FLAG

As a debugging aid, two statements are provided to trace the execution of program instructions.

TRON/TROFF (enable/disable trace flag)

Forms: TRON TROFF

The TRON/TROFF statements are used to trace the execution of program statements.

As an aid in debugging, the TRON statement (executed in either the Command or Indirect Mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

Example:

```
TRON
Ok
LIST
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINTJ;K;L
50 K=K+10
60 NEXT
70 END
Ok
RUN
[10] [20] [30] [40] 1 10 20
[50] [60] [30] [40] 2 20 30
[50] [60] [70]
Ok
TROFF
Ok
```

OVERLAY MANAGEMENT

BASIC-80 provides two statements, CHAIN and COMMON, which are useful for manipulating overlays. With these two statements, it is possible to merge several programs during the execution of a program, as well as pass several or all the variables to another program.

CHAIN (call overlay)

Form: CHAIN [MERGE] <filename>[,[<line number exp>] [,ALL][,DELETE<range>]]

The CHAIN statement is used to call a program and pass variables to it from the current program.

<filename> is the name of the program that is called.

Example:

```
CHAIN"PROG1"
```

<line number exp> is a line number or an expression that evaluates to a line number in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line.

Example:

```
CHAIN"PROG1",1000
```

<line number exp> is not affected by a RENUM command.

With the ALL option, every variable in the current program is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to specify the variables that are passed.

Example:

```
CHAIN"PROG1",1000,ALL
```

If the MERGE option is included, it allows a subroutine to be brought into the BASIC program as an overlay. That is, a MERGE operation is performed with the current program and the called program. The called program must be an ASCII file if it is to be MERGEd.

Example:

```
CHAIN MERGE"OVRLAY",1000
```

After an overlay is brought in, it is usually desirable to delete it so that a new overlay may be brought in. To do this, use the DELETE option. The line numbers in the <range> of the delete are affected by the RENUM command.

Example:

```
CHAIN MERGE"OVRLAY@",1000,DELETE 1000-5000
```

If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEFFN statement containing shared variables must be restated in the chained program.

COMMON (pass variables)

Form: COMMON<list of variables>

The COMMON statement is used to pass variables to a chained program.

The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though we recommend that they appear at the beginning. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending "( )" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Example:

```
100 COMMON A,B,C,D(),G$
 110 CHAIN "PROG3",10
          .
          .
          .
```

INSERT

Chapter Nine

Editing

OVERVIEW

In Edit Mode, it is possible to edit portions of a line without retyping the entire line. Upon entering Edit Mode, BASIC-80 types the line number of the line to be edited. Then it types a space and waits for the Edit Mode subcommand.

Edit Mode subcommands are used to insert, delete, replace, or search for text within a line. The subcommands are not echoed to the terminal. Some of the Edit Mode subcommands may be preceded by an integer which causes the command to be executed that number of times. When an integer is not specified, it is assumed to be one.

Edit Mode subcommands may be categorized according to the following functions:

1. Moving the cursor.
2. Inserting text.
3. Deleting text.
4. Finding text.
5. Replacing text.
6. Ending and restarting Edit Mode.

If BASIC-80 receives an unrecognizable command or illegal character while in Edit Mode, it sounds the bell (CTRL-G) and the command or character is ignored. You can invoke the Edit Mode by typing:

```
EDIT <line number>
```

Where <line number> is the number of the line to be edited. If no <line number> exists, an "Undefined line number" error will result.

The requested line number will be printed, followed by a space. The cursor will now be positioned to the left of the first character in the line.

Type in the following line:

```
100 FOR J = 1 TO 10:PRINT J:NEXT
```

This program line will be used to demonstrate the various Edit Mode commands.

MOVING THE CURSOR

n Space Bar

In Edit Mode, the Space Bar is used to move the cursor to the right. For example, using line 100 entered above, invoke the Edit Mode. The line number 100 should be displayed on your screen as such:

```
100 _
```

Now press the Space Bar. The cursor will move over one space. The first character of the program line will now be displayed. If this character was a blank, then a blank will be displayed on your screen. Keep pressing the Space Bar until the first non-blank character is displayed. At this point, the screen should look like this:

```
100 F_
```

It is also possible to move over more than one space at a time. Just type the number of spaces first, and then the Space Bar. For example, to move over five spaces, type 5 and then press the Space Bar once. The characters will be printed as you move over them.

```
100 FOR J=_
```

(Your display may not look exactly like this, as it depends on how may blanks you inserted in the program line.)

BACK SPACE

In Edit Mode, the BACK SPACE key moves the cursor one space to the left. The characters are not deleted as you move over them. To return to our example,

```
100 FOR J=_
```

if the cursor were positioned after the = sign, pressing BACKSPACE once should move the cursor under the = sign. Thus:

```
100 FOR J=
```

INSERTING TEXT

I (Insert)

The I command will insert text beginning at the current cursor position. The inserted characters are printed on the terminal. To terminate insertion, press the ESC key. If you press the RETURN during the insert command, the effect is the same as typing ESC and then RETURN.

Use the Space Bar to move over to the 0 in the 10.

```
100 FOR J=1 TO 10_
```

Now, suppose you want to change the 10 to 100. Press the I key (you don't have to terminate the entry with a RETURN). You are now in Insert Mode. To make the neccessary change, type a 0. The display should now look like this:

```
100 FOR J=1 TO 100_
```

Now that you have made the change, press the ESC key and you will exit Insert Mode. Now press the RETURN to save all your changes and return to BASIC-80 Command Mode. If you list line 100, it should look similiar to this:

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

During an insert command, you can use the BACK SPACE key on the terminal to delete characters on the left of the cursor.

If you try to insert a character that will make the line longer than 255 characters, a bell (CTRL-G) will be typed and the character will not be printed.

X (Extend Line)

The X command is used to extend the line. X moves the cursor to the end of a line. BASIC-80 then goes into the Insert Mode and allows text to be inserted as if an insert command had been given. When you are finished extending the line, type ESC or press RETURN and you will be returned to BASIC-80 Command Mode.

For example, to extend line number 100 you previously typed in, invoke Edit Mode with line number 100. The screen will show:

```
100 _
```

Now press the X key. The entire line will be displayed and the cursor will be at the end of the line:

```
100 FOR J=1 TO 100:PRINT J:NEXT_
```

Now you have been put into Insert Mode. You can now add another program statement to the end of this line. For example, type :PRINT"ALL DONE" and a RETURN. The line has now been extended to include this statement. If you were to LIST 100, it should look like this:

```
100 FOR J=1 TO 100:PRINT J:NEXT:PRINT"ALL DONE"
```

DELETING TEXT

nD (Delete)

nD deletes n characters to the right of the cursor. The deleted characters are echoed between backslashes, and the cursor is positioned to the right of the last character deleted. If there are fewer than n characters to the right of the cursor, the remainder of the line will be deleted.

For example, enter Edit Mode with line number 100 you previously typed in. Now, using the Space Bar, move the cursor over to the end of the FOR statement. The screen should look something like this:

```
100 FOR J=1 TO 100:_
```

Now type 8D. This will delete eight characters to the right of the cursor. The screen should look something like this:

```
100 FOR J=1 TO 100:\PRINT J:\
```

(Note that the characters deleted are enclosed in backslashes.)

Now press RETURN and you will be back to the BASIC-80 Command Mode. If you LIST 100, you should notice that the PRINT J statement has been deleted from the program line.

H (Hack and Insert)

H deletes all characters to the right of the cursor and then automatically enters Insert Mode. H is useful for replacing statements at the end of a line. For example, assume you wish to change the last statement of program line 100. First, you must enter Edit Mode with line number 100. Now move over to the NEXT statement with the Space Bar. The screen should look similiar to this:

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

Press the H key and then type STOP. Type a RETURN to save this change and exit to BASIC-80 Command Mode.

Now list line number 100. If you've been following the editing changes in this Chapter, the line should look like this:

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

FINDING TEXT

nS<ch>(Search)

The search subcommand searches for the nth occurence of <ch> and positions the cursor before it. The character at the current cursor position is not included in the search. If <ch> is not found, the cursor will stop at the end of the line. All characters passed over during the search are printed. NOTE: only characters to the right of the cursor are included in this search.

For example, using the current form of the sample line 100, enter Edit Mode with line 100. Next, type 2S: . This command will be used to search for the second occurrence of the colon character in program line 100. The display should look something like this:

```
100 FOR J=1 TO 100:NEXT_
```

At this point you can execute any command you wish. You could enter a counter variable after the NEXT statement by first entering Insert Mode and then typing a space and the variable J. Now hit ESC to exit Insert Mode. Finally, press RETURN in order to exit back to the BASIC-80 Command Mode. Now, if you were to list line number 100, it would look similar to this (assuming you followed the editing changes in this chapter):

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

nK<ch>(Search and "Kill")

The search and kill subcommand is similiar to the search subcommand except that all the characters passed over in the search are deleted. The cursor is positioned before <ch> and all the deleted characters are enclosed in backslashes.

For example, invoke the Edit Mode with the current version of line 100. Now type 2K:. This command will delete all of the characters in the line up to the second occurance of the colon. The screen should look similiar to this:

```
100 \FOR J=1 TO 100:NEXT J\_
```

The second colon still needs to be deleted, so type D. The screen should then look similiar to this:

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

Now press RETURN and LIST line 100. It should look like this:

```
100 STOP
```

REPLACING TEXT

nC(Change)

The change subcommand changes the specified number of characters beginning at the current cursor position. If you type only a C without a preceding number, the computer assumes that you wish to change only one character. If you enter a number n before you type C, then it assumes that you wish to change the next n characters.

After you have entered n characters, the Change Mode will be exited. If you attempt to enter any more characters, the bell is sounded (CTRL-G) and the extra characters are ignored.

For example, first retype the original line 100 as:

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

Next, enter Edit Mode with line 100. Your screen should look something like this:

```
100 _
```

Now let's assume that you want to change the terminal value in the FOR/NEXT loop from 100 to 150. You would have to move the cursor over to the first zero in 100. Use the Space Bar to move the cursor over. If you go too far, simply press the BACKSPACE key to move the cursor back.

```
100 FOR J=1 TO 1_
```

Now type C. BASIC-80 will assume that you wish to change only one character. Type 5 and then press RETURN. If you LIST 100, the new line should look like this:

```
100 FOR J=1 TO 150:PRINT J:NEXT
```

ENDING AND RESTARTING EDIT MODE

RETURN(Save changes and Exit)

If you press a RETURN, remainder of the line is printed, the changes you made are saved, and the computer returns to the BASIC-80 Command Mode.

E(Save Changes and Exit)

The E subcommand has the same effect as RETURN, except the remainder of the line is not printed.

Q(Cancel and Exit)

The Q subcommand returns to the BASIC-80 Command Mode without saving any of the changes that were made to the line during Edit Mode.

L(List Line)

The L subcommand lists the remainder of the line (saving any changes made so far) and repositions the cursor at the beginning of the line, still in the Edit Mode. L is usually used to list the line when you first enter Edit Mode. For example:

```
EDIT 100
100 _
<you type L>
<BASIC-80 responds:>
100 FOR J=1 TO 100:NEXT:STOP
100 _
```

A(Cancel and Restart)

The A subcommand lets you begin editing a line over again. It discards any changes made so far and restores the original line, repositioning the cursor at the beginning. In order to use the A subcommand,you must not be currently executing any other subcommand. If you are executing another command (such as Insert), press the ESC, and then press the A. In the following example, the operator first lists the original line, then makes changes in Insert Mode, then decides to start over, using the A subcommand to restore the original line:

```
EDIT 100
100 _
<operator types L>
100 FOR J=1 TO 100:NEXT:STOP
100 _
100 for J=1 TO 10_ <operator types I and adds a zero>
<operator types ESC>
<operator types L>
100 FOR J=1 TO 1000:NEXT:STOP
100 _
<operator types A>
100 _
<operator types L; note how original line has been restored>
100 FOR J=1 TO 100:NEXT:STOP
100 _
```

OTHER EDIT MODE FEATURES

SYNTAX ERRORS

When it finds a syntax error during the execution of a program, BASIC-80 will automatically enter Edit Mode at the line that caused the error. For example:

```
10 K = 2(4)
RUN
Syntax Error in 10
Ok
10 _
```

When you finish editing the line and press RETURN (or the E subcommand), BASIC-80 reinserts the line. This causes all variable values to be lost and all open files to be closed. To preserve the variable values for examination, first exit Edit Mode with the Q subcommand. BASIC-80 will return to the Command Mode and all variable values will be preserved.

CTRL-A

To enter the Edit Mode on the line you are currently typing, type CTRL-A. BASIC-80 will respond with a carriage return, an exclamation point, (!) and a space. The cursor will then be positioned at the first character in the line. At this point you may proceed by typing any Edit Mode subcommand.

CURRENT LINE EDITING

You may use the period (.) to denote the current line when you invoke the Edit Mode. So, the command:

```
EDIT .
```

will invoke the Edit Mode at the current line. The line number symbol (.) always refers to the current line.

INSERT

Chapter Ten

BASIC-80 Disk File Operations

OVERVIEW

BASIC-80 provides several sets of statements for creating and manipulating program and data files.

The file manipulation commands are very useful for manipulating program files. Some of these commands can also be used with data files.

The file management statements are used to open and close data files, check for end-of-file, and to obtain information about the size of a file.

The sequential access statements are used to access sequential files. The sequential access file is easy to use, but the data must be accessed sequentially.

The random access statements are used to access and manipulate random access files. The random access file requires more program steps than the sequential access, but the records in the file can be read in any order.

FILE MANIPULATION COMMANDS

This is a review of the commands and statements that are useful for manipulating program and data files. These statements and commands are also discussed in Chapter Three, "Command Mode Statements".

FILES ["<filename>"]

The FILES command lists the names of the files that are residing on the current disk. If the optional <filename> string is included, the names of the files on any specified disk can be listed.

KILL "filename"

The KILL command deletes the file from the disk. "filename" may be a program file, or a sequential or random access data file. If "filename" is a data file, it must be closed before it is killed.

LOAD "filename"[,R]

The LOAD command loads the program from disk into memory. The option R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before LOADing. If R is included, however, open data files are kept open. Thus programs can be chained or loaded in sections and can access the same data files.

MERGE "filename"

The MERGE command loads the program from disk into memory but does not delete the current contents of memory. The program line numbers on disk are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the "merged" program resides in memory and BASIC-80 returns to Command Mode.

NAME "oldfile" AS "newfile"

To change the name of a disk file, execute the NAME statement, NAME "oldfile" AS "newfile". NAME may be used with program files, random files, or sequential files.

RESET

RESET reads the directory information off of a newly inserted disk which you have exchanged for the disk in the current default drive. RESET does not close files that were opened on the former default disk. Therefore, use RESET only after you have closed any open files and replaced the current default disk.

RUN "filename"[,R]

RUN "filename" loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open.

SAVE "filename"[,A]

The SAVE command writes to disk the program that is currently residing in memory. Thoption writes the program in ASCII format. (Otherwise, BASIC uses a compressed binary format.)

Protected Files

If you wish to save a program in an encoded binary format, use the "Protect" option with the SAVE command. For example:

```
SAVE "MYPROG",P
```

A program saved this way cannot be listed or edited.

FILE MANAGEMENT STATEMENTS

BASIC-80 provides a full set of I/O statements to be used for disk file management. These statements are listed below:

Statement	Function
OPEN	Opens a disk file and assigns a file number to the disk file.
CLOSE	Closes a disk file and de-assigns the file number from the disk file.
EOF	Returns −1 (true) if the end of a file has been reached.
LOF	Returns the number of records present in the last extent accessed.
LOC	Returns the next record to be accessed for a random file and the total number of sectors accessed for a sequential file.

Table 10-1
File Management Statements.

The OPEN statement is used to assign a file number to a disk file name. Also, the OPEN statement is used to define the mode in which the file is to be used (sequential or random access).

The CLOSE statement performs the opposite function of the OPEN statement. It will de-assign the file number from a disk file name.

The EOF function will return −1 (true) if the end of a sequential file has been reached. The EOF function can also be used with random files to determine the last record number.

The LOF function will return the number of records present in the last extent accessed.

The LOC function, when used with a random file, will return the next record to be accessed. When used with a sequential file,it returns the number of records accessed since the file was opened.

These statements are discussed on the following pages. For a detailed programming example that utilizes these statements, see "Appendix F."

OPEN (open disk data file)

Form: OPEN "mode",[#]<filenumber>,"<filename>",[,<,reclen>]

where:

"mode" is a string expression whose first character is one of the following mode specification strings:

O	Specifies sequential output mode.
I	Specifies sequential input mode.
R	Specifies random input/output mode.

This string expression will be referred to as the "mode string".

<filenumber> is an integer expression which represents the file number associated with the file. This number will be used in subsequent I/O operations.

<filenumber> must not exceed the number of files that were set during the BASIC-80 initialization process. If no files were set during the initilization process, BASIC-80 will assume a maximum of 3. (See Chapter One, "System Introduction & General Information", for more information about this initialization process.)

"<filename>" is the fully qualified CP/M file name. No extensions are assumed, so the file name must include this information. If no drive is specified, the current default drive is assumed.

<reclen> is an integer expression which, if included, sets the record length for random files. The maximum record length is 256 bytes. The default record length is 128 bytes. If a record length greater than 128 bytes is desired, this length must also be specified when BASIC-80 is initialized. This record length option can only be used with random files. Any attempt to declare the size of a sequential record will result in a "Syntax error".

The OPEN statement is used to associate a file number with a file name. The OPEN statement also defines the mode in which the file will be used (sequential or random access). Subsequent I/O operations will reference the file number assigned to a file name. For example, assume that a file was opened using the following statement:

```
OPEN "I",2,"SAMPLE.DAT"
```

This statement will assign file number 2 to the file SAMPLE.DAT. Because no drive name was specified, BASIC-80 will assume that SAMPLE.DAT resides on the current default drive. The mode string for this file specifies "I" -- sequential input.

If SAMPLE.DAT does not exist on the current default disk, an error will be generated, since input can not be performed on a non-existant file. Now, to input data from this file, the following statement would be used:

```
INPUT#2,<variable list>
```

Note that this INPUT# statement references file number 2, and file number 2 was the number assigned to the file SAMPLE.DAT. (This is only a general form of the INPUT# statement. A detailed discussion of the INPUT# statement appears later in this Chapter.)

Now assume that the following OPEN statement is used:

```
OPEN "O",3,"B:OUTPUT.DAT"
```

This will assign file number 3 to the file OUTPUT.DAT. Since the file name does contain the drive specification B:, BASIC-80 will create this output file on drive B:. If this file already exists on drive B:, it will be destroyed, and all previous contents of the file will be lost. Now, to output data to this file, the following statement would be used:

```
WRITE#3,<variable list>
```

The WRITE# statement references file number 3, and file number 3 had been previously assigned to the file B:OUTPUT.DAT. So, the data specified in the <variable list> would be written to the file B:OUTPUT.DAT. (The WRITE# statement is discussed in more detail later in this Chapter.) A file can also be opened for random I/O. One OPEN statement can be used to open the file for both random input and random output. For example, the following statement will open a file for random I/O.

```
OPEN "R",1,"RANDOM.DAT"
```

The file, RANDOM.DAT, is opened for random I/O. If RANDOM.DAT does not exist, it will be created on the current default disk. Now, either random input or random output can be performed with this file. Note that no record size was specified with this OPEN statement. Therefore, BASIC-80 will assume the default record size of 128 bytes. A different record size can be specified with the OPEN statement. (But only for a random access file.)

For example, to open the file RANDOM.DAT for random access, and declare a record size of 32 bytes, the following statement would be used:

```
OPEN "R",1,"RANDOM.DAT",32
```

Now the record size would be 32 bytes. The CP/M sector size is 128 bytes. Therefore, four records would be stored in each CP/M sector. The record size can also be set during the initialization procedure with the /S switch. (See Chapter 1, "System Introduction & General Information," for the initialization procedure.)

It is important to note that the mode a file was opened under must be with the mode in which the file is accessed. For example, consider the following statement:

```
OPEN "I",1,"TEST.DAT"
```

The file TEST.DAT has been opened for sequential input and assigned to file number 1. Now an attempt to perform ouput on this file would be invalid and would generate an error message. For example:

```
WRITE#1,"HELLO THERE"
```

This WRITE# statement references file number 1. The previously executed OPEN statement has set the mode for file number 1 as sequential input. So this WRITE# would be invalid and would generate an error message.

However, there is an exception to this rule. Under certain circumstances several sequential I/O statements may be used with a random file. The conditions for using these sequential I/O statements with random files are explained in the last part of this chapter.

CLOSE (close disk data file)

Form: CLOSE [#] [<filenumber>]

The CLOSE statement is used to conclude I/O activity to a disk data file.

<filenumber> is the number under which the file was opened. A CLOSE with no arguments will close all open files.

Assume the following OPEN statement appears in a program:

```
OPEN "O",1,"ARTIST.DAT"
```

Now a sequential output statement may reference this file. When output to this file has concluded, it should be closed with the CLOSE statement.

```
CLOSE #1
```

This statement will disassociate file number 1 from the file ARTIST.DAT. Any reference to file number 1 would now be invalid. The file may then be reopened using the same or a different file number. For example:

```
OPEN "I",3,"ARTIST.DAT"
```

The file ARTIST.DAT is now associated with file number 3, and is opened for sequential input. Now, a sequential input operation with this file would be valid. When the input operation has concluded, this file should be closed with the CLOSE statement.

```
CLOSE #3
```

The file could again be reopened:

```
OPEN "R",3,"ARTIST.DAT"
```

The file number 3 has again been associated with the file ARTIST.DAT, but, this time the file has been opened for random I/O.

A CLOSE for a sequential output file writes the final buffer of output to the disk file. (This subject is covered in more detail later in this chapter.)

The END statement and the NEW command will close all disk files automatically. Any attempt to edit or modify a program will also automatically close all open disk files. (The STOP statement does not close disk files.)

EOF (check for end-of-file)

Form: EOF(<filenumber>)

<filenumber> is the file number assigned to a disk data file in a previously executed OPEN statement.

The EOF function will return -1 (true) if the end of a sequential file has been reached.

The EOF is useful for detecting when the end of a sequential file has been reached. The EOF function should be used in conjunction with the INPUT# statement and the LINE INPUT# statement to avoid "Input past end" errors.

The EOF function may also be used with random files. If a GET is done past the end of the random file, the EOF function will return −1 (true). This may be used to find the size of a random file.

Example:

```
10 OPEN "I",1,"DATA"
20 IF EOF(1) THEN 100
30 INPUT#1,A$
40 GOTO 20
   .
   .
   .
100 PRINT "END-OF-FILE REACHED"
```

LOF (return number of records)

Form: LOF(<filenumber>)

<filenumber> is the file number assigned to a disk data file in a previously executed OPEN statement.

The LOF function returns the number of records present in the last extent that was accessed. If the file does not exceed one extent, then LOF returns the true length of the file. (Refer to the "CP/M Application Programmer's Manual" for more information on extents.)

Example:

```
110 IF NUM% > LOF(1) THEN PRINT "INVALID ENTRY"
```

LOC (return record number)

Form: LOC(<filenumber>)

<filenumber> is the file number assigned to a disk data file in a previously executed OPEN statement.

When used with a random file, the LOC function returns the current record number. The current record is the record number one greater than that of the last record accessed. The first time a particular file is accessed, the current record is 1. The largest possible record number is 32767.

When used with a sequential file, the LOC function returns the number of sectors (128 byte blocks) accessed since the file was opened.

Examples:

```
10 OPEN "I",1,"TEST.DAT"
20 OPEN "R",2,"RANDOM.DAT"
          .
          .
          .
200 PRINT"SECTORS READ--";LOC(1)
210 PRINT"NEXT REC#--";LOC(2)
```

BASIC-80 SEQUENTIAL I/O

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes to accessing the data. The data that is written to a sequential file is stored, one item after another (sequentially), in the order it is sent and it must be read back in the same order. The data is stored as a stream of ASCII characters.

Sequential Access Statements

INPUT#	Input data from sequential file.
LINE INPUT#	Input entire line from sequential file.
PRINT# PRINT# USING	Write data to sequential file.
WRITE#	Write data to sequential file (with delimiters automatically inserted).

Table 10-2
Sequential Access Statement.

INPUT# (input data from sequential file)

Form: INPUT#<filenumber>,<variable list>

The INPUT# statement is used to read data items from a sequential disk file and assign them to program variables. The data will be read sequentially. When the file is opened, a pointer will be set to the beginning of the file. Each time data is read from the file, the pointer will advance. To start reading over from the beginning of a file, the sequential file must be closed and re-opened.

<filenumber> is the number used when the file was opened for input. <variable list> contains the variable names that the input data will be assigned to. (The variable data type must match the type specified by the variable name. It is invalid to read a string data value into a numeric variable.)

Numeric Input

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces are ignored.

The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed or comma.

For example, assume the following data image exists on a disk file:

(note: the b represents a blank or space - ASCII 32)

```
bb2.1234b-123.234bb456<carriage return>
```

Then the INPUT statement:

```
INPUT#1,X,Y,Z
```

or the sequence of INPUT statements:

```
INPUT#1,X:INPUT#1,Y:INPUT#1,Z
```

will assign the data values as follows:

```
X=2.1234
Y=-123.234
Z=456
```

The following discussion assumes the image on the disk is (note: the b represents a blank or space - ASCII 32):

```
bb2.1234b−123.234bb,456<carriage return>
```

And the INPUT statement used to access the data is:

```
INPUT#1,X,Y,Z
```

The two blanks before the value 2.1234 are leading spaces; therefore, they are ignored. The next character encountered is a 2, and this is considered the start of the first numeric field.

The BASIC-80 I/O processor now scans for the terminator of the first numeric field. The blank between 2.1234 and −123.234 is this terminator. So when BASIC-80 encounters this blank, it assumes that the first numeric field has ended. This first numeric field is assigned to the first item in the variable list, the variable X.

The BASIC-80 I/O processor now scans for the beginning of the second numeric field. The minus sign (−) is considered the start of the second numeric field. The BASIC-80 I/O processor will scan for the terminator of the second numeric field. The comma between −123.234 and 456 is this terminator. So, when BASIC-80 encounters this comma, it assumes that the second numeric field has ended. This second numeric field is assigned to the second item in the variable list, the variable Y.

The BASIC-80 I/O processor now scans for the beginning of the third numeric field. The number 4 is considered the start of the third numeric field. The BASIC-80 I/O processor will then scan for the terminator of the third numeric field. The carriage return after 456 is this terminator. So when BASIC-80 encounters this carriage return, it assumes that the third numeric field has ended. This third numeric field is assigned to the third item in the variable list, the variable Z.

At this point, all three variables in the variable list have values assigned to them, so execution of the INPUT statement has been completed. Execution continues with the next statement.

String Input

When BASIC-80 scans the sequential data file for a string item, leading spaces, carriage returns, and line feeds are ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item.

This string is considered an unquoted string, and will terminate on a comma, carriage return or line feed (or after 255 characters have been read).

If this first character is a quotation mark, the string is considered a quoted string. The string item will consist of all characters read between the first quotation mark and the next quotation mark. Commas, blanks, and carriage return characters can be included in this string. A quoted string may not contain a quotation mark within the quoted string.

For example, assume the following data image exists on a disk file:

(b represents a blank or space -- ASCII 32)

```
BENTON,HARBOR,MI"49022"<carriage return>
```

Then the statement:

```
INPUT#1,A$,B$,C$
```

would assign the data values as follows:

```
A$=BENTON
B$=HARBOR
C$=MI"49022"
```

Note that the comma is used as the terminator in the above example. All three strings are considered to be unquoted strings.

In the last string field, the quotation mark is considered as part of the string. This is because the string starts with the letter M and is terminated by a carriage return.

Assume a comma is inserted between MI and "49022". The disk image would then look like this:

```
BENTON,HARBOR,MI,"49022"
```

Now there are a total of four string fields. The first three are unquoted strings fields, and the last is a quoted string field. These four fields could be input with the following statement:

```
INPUT #1,A$,B$,C$,D$
```

the variable values would be assigned as follows:

```
A$=BENTON
B$=HARBOR
C$=MI
D$=49022
```

The variable D$ would not contain the quotation marks because the quotation marks were used to terminate the field, and as such they do not represent data values.

LINE INPUT# (input entire line from sequential file)

Form: LINE INPUT#<filenumber>,<string variable>

The LINE INPUT# statement is used to read an entire line (up to 255 characters), without delimiters, from a sequential disk data file to a string variable.

<filenumber> is the file number assigned to the file with the OPEN statement. The file must be opened for sequential input (I mode). <variable> is the variable name to which the input will be assigned.

LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/line feed sequence, and the next LINE INPUT# reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is preserved.)

If no carriage return is found, LINE INPUT# will read until 255 characters have been read. These 255 characters will then be assigned to the string variable.

LINE INPUT# is especially useful if each field of a data file has been terminated with a carriage return, or if a BASIC-80 program saved in ASCII mode is being read as data by another program.

For example, assume the following program exists in a disk file:

```
10 OPEN "O",1,"LIST" <carriage return>
20 INPUT C$ <carriage return>
30 PRINT #1, C$ <carriage return>
40 CLOSE #1 <carriage return>
```

then the statement:

```
LINE INPUT#1,Z$
```

could be repetitively used to read each program line, one line at a time.

PRINT# AND PRINT# USING (write to sequential disk file)

Forms:

PRINT#<filenumber>,<list of expressions>

PRINT#<filenumber>,USING<string exp>;<list of expressions>

The PRINT# statement is used to write data to a sequential disk file.

<filenumber> is the number used when the file was opened for output. The expressions in <list of expression> are the numeric and/or string expressions that will be written to the file.

PRINT# does not compress data on the disk. An image of the data is written to the disk, just as it would be displayed on the terminal with a PRINT statement. (The PRINT statement is discussed in Chapter Four, "Program Statements.") For this reason, take care to delimit the data on the disk so it will be input correctly from the disk.

In the list of expressions, numeric expressions should be delimited by semicolons.

For example:

```
    PRINT#1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to disk.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the disk, use explicit delimiters in the list of expressions.

For example, let `A$="CAMERA"` and `B$="93604-1"`.

The statement:

```
      PRINT#1,A$;B$
```

would write `CAMERA93604-1` to the disk. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

```
      PRINT#1,A$;",";B$
```

The image written to disk is:

```
CAMERA,93604-1
```

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to disk surrounded by explicit quotation marks, CHR$(34).

For example, let `A$="CAMERA, AUTOMATIC"` and `B$=" 93604-1"`. The statement:

```
PRINT#1,A$;B$
```

would write the following image to disk:

```
CAMERA, AUTOMATIC 93604-1
```

and the statement:

```
INPUT#1,A$,B$
```

would input `"CAMERA"` to `A$` and `"AUTOMATIC 93604-1"` to `B$`. To separate these strings properly on the disk, write double quotes to the disk image using `CHR$(34)`.

The statement:

```
PRINT#1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image to disk:

```
"CAMERA, AUTOMATIC" "    93604-1"
```

and the statement:

```
INPUT#1,A$,B$
```

would input `"CAMERA, AUTOMATIC"` to `A$` and `" 93604-1"` to `B$`.

The PRINT# statement may also be used with the USING option to control the format of the disk file. For example:

```
PRINT#1,USING"$$###.##,";J;K;L
```

The comma at the end of the format string serves to separate the items in the disk file. (For a complete discussion of the PRINT USING statement, refer to Chapter Eight, "Special Features.")

NOTE: The WRITE# statement will automatically insert the proper delimiters between data items in a sequential file.

WRITE#(write to sequential disk file)

Form: WRITE#<filenumber>,<list of expressions>

The WRITE# statement is used to write data to a sequential file.

<filenumber> is the number which was assigned to the file with an OPEN statement. The file must be open for sequential output (O mode). The expressions in the list are string or numeric expressions, and they must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to disk and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/line feed sequence is inserted after the last item in the variable list is written to the disk file.

Example: Let `A$="CAMERA"` and `B$="93604-1"`. The statement:

```
WRITE#1,A$,B$
```

writes the following image to disk:

```
"CAMERA","93604-1"
```

A subsequent INPUT# statement, such as:

```
INPUT#1,A$,B$
```

would input `"CAMERA"` to `A$` and `"93604-1"` to `B$`.

Note: The WRITE# statement is recommended for most applications using sequential output. Most problems arising from using sequential files are a result of not inserting the proper delimiters between data items. The WRITE# statement eliminates the need to be concerned with delimiting data items, thus eliminating most problems associated with sequential I/O.

In those cases where the WRITE# statement will not provide the flexibility needed for some unique sequential output application, use of the PRINT# or PRINT# USING statement should be considered. Care should be taken to insure that all the data items are separated by the proper delimiters.

Sequential Access Techniques

CREATING AND ACCESSING A SEQUENTIAL FILE

The following program steps are required to create a sequential file and access the data in the file:

Open the file for sequential output.

```
OPEN "O",#1,"DATA.DAT"
```

This step will associate the file number 1 with the file DATA.DAT. Because the O mode string was specified, the file will be opened for sequential output. Since no drive specification was included with the file name, the current default drive will be assumed.

If a file DATA.DAT already exists on the current default drive, contents of this file will be lost. This is due to the fact that, when a file is opened for sequential output, the BASIC-80 I/O processor will move the EOF marker to the beginning of the file. Thus, the previous contents of the file can no longer be accessed.

Write data to the file

```
WRITE#1,A$,B$,C$
```

This step assumes that some string value has been assigned to the string variables A$,B$ and C$. The WRITE# statement will write data to the file with delimiters, so it is not neccessary to insert any delimiters.

The PRINT# statement could have been used to write the data to this sequential file, but then it would have been necessary to insert delimiters between the data items. So for most applications using sequential output, it is more efficient to use the WRITE# statement.

Close the file

```
CLOSE#1
```

This statement will write any remaining data from the buffer to the disk file. Output to this file will then be terminated. The file must be closed before it can be reopened for sequential input.

Reopen the file for input

```
OPEN "I",#1,"DATA.DAT"
```

The file number 1 is again associated with the file DATA.DAT. This time, the file is opened for sequential input.

Read the data

```
INPUT#1,X$,Y$,Z$
```

The data will be read from the file DATA.DAT and assigned to the string variables X$,Y$ and Z$

NOTE: The above example ignores the role of the I/O buffer in the sequential I/O process. Actually, BASIC-80 reads and writes in 128-byte blocks. So each INPUT# or WRITE# statement may not necessarily require a disk access.

With sequential output, each WRITE# or PRINT# will place the data in the buffer area. When the buffer is filled with data, the data will actually be written to the disk file.

With sequential input, 128 bytes will be read and placed in the buffer area. Then the BASIC-80 I/O processor will sort through the data in the buffer to satisfy the INPUT# statement variable list.

ADDING DATA TO A SEQUENTIAL FILE

As soon as an existing sequential file is opened for output ("O" mode) ,the current contents of the file are destroyed. Thus, several program steps are required to add data to an exisiting sequential file. The following procedure can be used to add data to an existing file called "DATA.DAT"

Open "DATA.DAT" for sequential input

```
OPEN "I",1,"DATA.DAT"
```

This step associates file number 1 with the data file DATA.DAT. This file will be opened for sequential input. Since no drive specification was included with the file name, BASIC-80 will assume the current drive. If the file DATA.DAT can not be found on the current default drive, a "File not found" error will be generated.

Open a second file called "TEMP.TMP" for sequential output

```
OPEN "O",2,"TEMP.TMP"
```

The file, TEMP.TMP will be used as a temporary work file. After this process is completed, this file will be renamed and it will contain the original data as well as the newly created data.

Read in the data in "DATA.DAT" and write it to "TEMP.TMP"

```
INPUT#1,A$,B$,C$
WRITE#2,A$,B$,C$
```

This step must be repeatedly executed until all the data in file #1 is read.

Close "DATA.DAT" and kill it.

```
CLOSE#1
KILL"DATA.DAT"
```

This file is no longer needed, as the information from this file has been copied into the file TEMP.TMP

Write the new information to "TEMP.TMP"

```
WRITE#2,A$,B$,C$
```

The data assigned to the string variables A$,B$ and C$ will be written to the disk file.

Close the file

```
CLOSE#2
```

This step will terminate the output operation performed with this file.

Rename "TEMP.TMP" as "DATA.DAT"

```
NAME "TEMP.TMP" AS "DATA.DAT"
```

Now there is a file on disk called "DATA.DAT" that includes all the previous data plus the new data that was added to the file.

BASIC-80 RANDOM I/O

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. One advantage is that random files require less room on the disk because BASIC-80 stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to random files is that data can be accessed randomly, i.e., anywhere on the disk — it is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in distinct units called records and each record is numbered.

All data stored in a random file must be a string data type.

To store numeric values in a random file, the numeric values must be converted to strings. Several functions have been provided to convert numeric values to strings. These functions, (MKI$,MKS$,MKD$), are explained later in this Chapter.

Random Access Statements

Statement	Function
FIELD	Set up random file buffer.
LSET	Move data to random buffer. (left-justified)
RSET	Move data to random buffer. (right-justified)
GET	Read random record.
PUT	Write random record.
MKI$	Make integer into 2-byte string.
MKS$	Make single-precision number into 4-byte string.
MKD$	Make double-precision number into 8-byte string.
CVI	Convert 2-byte string to integer.
CVS	Convert 4-byte string to single-precision number.
CVD	Convert 8-byte string to double-precision number.

Table 10-3
Random Access Statements.

FIELD (set up random file buffer)

Form:

FIELD#<filenumber>,<field width> AS <string variable>

The FIELD statement is used to allocate space for variables in a random file buffer.

<filenumber> is the number assigned to the random file in the OPEN statement. <field width> is the number of characters (bytes) to be allocated to <string variable>.

For example:

```
FIELD#1, 20 AS N$, 10 AS ID$, 40 AS ADD$
```

allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does **not** place any data in the random file buffer, but instead defines the fields in the random file buffer.

A FIELD statement can only reference a file which has been opened for random I/O (R mode). The FIELD statement must also be executed prior to performing any I/O operation with the random file.

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was opened. Otherwise, a "Field overflow" error occurs. (The default record length is 128.)

If a number smaller than 128 is specified for the record length, the BASIC-80 I/O processor will take care of blocking and deblocking the record. For example, if a record length of 32 bytes is specified in the OPEN statement, the BASIC-80 I/O processor will block 4 of these logical records per physical record (sector). The user program is not responsible for blocking and deblocking these logical records.

If a number greater than 128 is specified for the record length, the BASIC-80 will also take care of blocking and deblocking the record. This number must be specified by using the /S switch when initializing BASIC-80. The largest record size allowed is 256 bytes.

With previous versions of Microsoft BASIC, the user program did have to assume responsibility for blocking and deblocking records.

Any number of FIELD statements may be executed for the same file, and all FIELD statements that have been executed are in effect at the same time. For example, the following FIELD statement could be used to define a 32-byte random buffer:

```
FIELD#1, 16 AS F1$,16 AS F2$
```

This FIELD statement would allocate the first 16 characters (bytes) of the random buffer to the variable F1$ and the next 16 characters (bytes) to the variable F2$. Then, another FIELD statement could be used to redefine the buffer:

```
FIELD#1,32 as BUFF$
```

So the variable BUFF$ would refer to all 32 characters in the buffer. F1$ would still refer to the first 16 characters and F2$ would still refer to the second 16 characters.

Do **not** use a fielded variable name in an INPUT or LET statement. Once a variable name is fielded, it points to a specific address in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

Examples:

```
FIELD#1,128 AS IBUFF$

FIELD#4,10 AS A$(1),10 AS A$(2),10 AS A$(3)

FIELD#2,I AS STUFF$
```

(Note: the variable I must be assigned an integer value prior to the execution of this statement.)

LSET/RSET (move data to random buffer)

Forms: LSET <fielded variable> = <string expression>

RSET <fielded variable> = <string expression>

The LSET/RSET statements are special assignment statements used to assign a string expression to a variable that has appeared in a FIELD statement (fielded variable).

The LSET/RSET statements are used to move data from memory to a random file buffer. This step is performed in preparation for a PUT statement. The only way to move data to a random buffer is by using the LSET/RSET statement.

If the <string expression> requires fewer bytes than were fielded to the <fielded variable>, LSET left-justifies the string in the field by adding spaces on the right. RSET is used to right-justify the string in the field by adding spaces on the left.

The only difference between LSET and RSET is the fact that LSET left-justifies the field and RSET right-justifies the field. If the string is too long for the field, characters are dropped from the right.

Numeric values must be converted to strings before they are LSET or RSET. Several special random I/O functions have been provided to perform this conversion. (Refer to the discussion of the MKI$, MKS$, and the MKD$ functions later in this Chapter.)

Examples:

```
150 LSET A$=MKS$(AMT)
160 LSET D$=DESC$
170 LSET V$="LEFT-JUSTIFY AND PLACE IN BUFFER"
180 RSET G$="RIGHT-JUSTIFY AND PLACE IN BUFFER"
```

String variables A$,D$,V$ and G$ must have appeared in a previously executed FIELD statement.

GET (read random record)

Form: GET [#]<filenumber>[,<record number>]

The GET statement is used to read a record from a random disk file into a random buffer. Before executing a GET statement, the file to be accessed must be opened for random I/O.

Additionally, the random file buffer must have been defined with a FIELD statement. If the random file buffer has not been defined, there will be no way to access the data after the GET has been executed.

<filenumber> is the number under which the file was opened. If <record number> is omitted, the current record is read into the buffer. The current record is the record number one greater than that of the last record accessed. The first time a particular file is accessed, the current record is 1. The largest possible record number is 32767.

If an attempt is made to GET a record whose number is higher than that of the last record number in the file, the buffer will be filled with NUL characters (ASCII Ø), although no error will be generated. The LOF function can be used to prevent this from occurring.

Examples:

```
GET#1,100

GET#2

GET FILE,IREC

GET#5,REC
```

PUT (write random record)

Form: PUT [#]<filenumber>[,<record number>]

The PUT statement is used to write a record from a random buffer to a random disk file. Before executing a PUT statement, the file to be accessed must be opened for random I/O.

Additionally, the random file buffer must have been defined with a FIELD statement. If the random file buffer has not been defined, there will be no way to move data into the buffer before executing the PUT statement.

<filenumber> is the number under which the file was opened. If <record number> is omitted, the current record is written. The current record is the record number one greater than that of the last record accessed. The first time a particular file is accessed, the current record is 1. The largest possible record number is 32767.

If the <record number> is higher than the end-of-file record number, <record number> becomes the new end-of-file record number. Space will be allocated on the disk to accommodate the new end-of-file record, as well as all lower numbered records.

Before executing a PUT statement, the data to be written to a disk file must be moved into the buffer area. The LSET/RSET statements are used to move the data to the random file buffer.

Examples:

```
PUT#1

PUT#2,43

PUT I,J-1

PUT I,4
```

MKI$, MKS$, MKD$ **(make a numeric value into a string)**

Forms: MKI$(<integer expression>)
MKS$(<single-precision expression>)
MKD$(<double-precision expression>)

The "make" functions, (MKI$, MKS$, MKD$) are used to convert numeric value to string value. Any numeric value that is placed in a random file buffer must be converted to a string.

The MKI$ function is used to convert an integer to a 2-byte string. The integer expression must be in the allowable range for integer values. If it is not, an "Illegal function call" error will be generated. Any fractional portion of the number will be truncated.

The MKS$ function is used to convert a single-precision number to a 4-byte string. The MKD$ function is used to convert a double-precision number to an 8-byte string.

These functions will not move the data to the random buffer. So after a numeric value is converted to a string, it still must be moved to the random file buffer. Additionally, the random file buffer must have been defined with a FIELD statement.

If the random file buffer has not been defined, there will be no way to access the data after the GET has been executed. The data must also be moved into the random buffer using LSET or RSET.

For example, to convert the integer variable IV% to a string and assign it to the field variable FV$, the following single program statement could be used:

```
LSET FV$ = MKI$(IV%)
```

The variable FV$ should have appeared in a previously executed FIELD statement.

Example:

```
90 AMT=(K+T)
100 FIELD #1, 8 AS D$, 20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ = A$
130 PUT #1
```

CVI, CVS, CVD (Converting string to numeric form)

Forms:
CVI (<2-byte string>)
CVS (<4-byte string>)
CVD (<8-byte string>)

The CVI, CVS and CVD functions are used to convert string values to numeric values. These functions are generally used to convert numeric values that have been read from a random disk file. Data is always stored in random files as a string data type. Therefore, a numeric value read from a random disk file must be converted from a string back into a number.

The CVI function converts a 2-byte string to an integer. If the length of the string is greater than 2 bytes, only the first two characters in the string will be used. If the length of the string is less than 2 bytes, an "Illegal function call" error will result.

The CVS function converts a 4-byte string to a single-precision number. If the length of the string is greater than four bytes, only the first four characters in the string will be used. If the length of the string is less than four bytes, an "Illegal function call" error will result.

The CVD function converts an 8-byte string to a double-precision number. If the length of the string is greater than eight bytes, only the first eight characters in the string will be used. If the length of the string is less than eight bytes, an "Illegal function call" error will result.

Example:

```
PRINT CVS(A$)

A#=CVD(BUFF$)

I = I+CVI(I$)
```

Random Access Techniques

CREATING A RANDOM ACCESS FILE

The following program steps are required to create a random file.

OPEN the file for random access

```
OPEN "R", 1 "FILE.DAT",32
```

In this example, the mode string specifies "R"—random access. File number 1 is assigned to the file FILE.DAT. Since no drive specification was included with this file name, the current default drive is assumed. This example also specifies a record length of 32 characters (bytes). If the record length is omitted, the default record length is 128 characters (bytes).

Set up the random file buffer

```
FIELD#1, 20 AS NAME$, 4 AS A$, 8 AS P$
```

Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file. The FIELD statement references file number 1, which has been opened for random input. (It is invalid to FIELD a file which has been opened for sequential input or output.)

This FIELD statement will allocate the first 20 characters of the random file buffer for the variable NAME$, the next four characters for the variable A$, and the next eight characters for the variable P$.

Move the data into the random buffer

```
LSET NAME$=X$
LSET A$=MKS$(AMT)
LSET P$=TEL$
```

Use LSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ for a single-precision value, and MKD$ for a double-precision value.

In this program step, the single-precision variable AMT is first coverted to a string, and then it is assigned to the variable A$. The variable A$ has appeared in a previous FIELD statement. The FIELD statement was used to allocate four characters (bytes) to the variable A$.

Write data to disk

```
PUT#1
```

Write the data from the buffer to the disk using the PUT statement. No record number was specified with this PUT statement, so the current record number will be written. The current record is the record number one higher than the last record accessed. The first time a file is accessed, the current record is one.

Do not use a fielded string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

ACCESSING A RANDOM ACCESS FILE

The following program steps are required to access a random file:

OPEN the file for random access

```
OPEN "R",#1,"FILE.DAT",32
```

This step will open the file "FILE.DAT" for random access. The file can now be accessed by referring to file number 1.

Set up random file buffer

```
FIELD#1, 20 AS NAME$,4 as A$, 8 AS P$
```

Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file. In this example, 20 characters (bytes) are allocated to the string variable NAME$, four characters are allocated to the string variable A$, and eight characters are allocated to the string variable P$.

NOTE: In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

Read data into buffer

```
GET#1
```

Use the GET statement to move the desired record into the random buffer. No record number was specified with this GET statement, so the current record number will be read. The current record is the record number one higher than the last record accessed. The first time a file is accessed, the current record is one.

Access data in the buffer

The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single-precision values, and CVD for double-precision

```
PRINT NAME$
AV=CVS(A$)
DP#=CVD(P$)
```

Additional Features

After a GET statement, INPUT# and LINE INPUT# may be used to read characters from the random file buffer. PRINT#, PRINT# USING, and WRITE# may also be used to put characters in the random file buffer before a PUT statement.

In the case of WRITE#, BASIC-80 pads the buffer with spaces (if necessary) and then inserts a carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

INSERT

Chapter Eleven

Microsoft BASIC-80 Summary

OVERVIEW

This Chapter is a summary of the important concepts, ideas, keywords, etc. of the BASIC-80 programming language. The various intrinsic functions as well as the string functions are also included in this chapter.

Abbreviations

Abbreviation	Function
?	Use in place of PRINT.
'	Use in place of REM.
.	"current line";use in place of line number with LIST, EDIT, etc.

Data Type Declaration Characters

Character	Data Type	Examples
$	String	`ZDS$,WLW$`
%	Integer	`I%,VALUE%`
!	Single-Precision	`V!,FLAG!`
#	Double-Precision	`DP#,PL#`
D	Double-Precision (exponential notation)	`1.23456789D-12`
E	Single-Precision (exponential notation)	`1.23456E+23`

Arithmetic Operators

Operator	Operation Performed
+	Addition
-	Subtraction
*	Multiplication
/	Division (floating point)
\	Integer division
^	Exponentiation

String Operator

Operator	Operation Performed	Example
+	concatenate (string together)	"A"+"B"+"C"

Relational Operators

Operator	Numeric Expressions	String Expressions
<	Less than	Precedes
>	Greater than	Follows
=	Equal to	Equals
<= or =<	Less than or equal to	Precedes or equals
>= or =>	Greater than or equal to	Follows or equals
<> or ><	Does not equal	Does not equal

Logical Operators

Operator	Function
NOT	Bitwise negation
AND	Bitwise disjunction
OR	Bitwise conjunction
XOR	Bitwise exclusive OR
IMP	Bitwise implication
EQV	Bitwise equivalence

Commands

Command/Function	Examples
AUTO <line number>,<increment>	

Enable automatic line numbering starting at <line number> and incrementing by <increment>.	`AUTO`
	`AUTO 10`
	`AUTO 5,5`

CLEAR	
Set numeric values to zero, strings to null.	`CLEAR`
CLEAR,<expression>	
Same as CLEAR, but <expression> is used to set the high memory limit for use by BASIC-80.	`CLEAR ,32768`
CLEAR,<expression1>,<expression2>	
Same as CLEAR<expression> but <expression2> is used to set the amount of stack space for use by BASIC-80.	`CLEAR,32768,2000`
CONT	
Continues program execution after a BREAK or STOP.	`CONT`
DELETE <line number>	
Deletes the specified line number in the current program.	`DELETE 100`
DELETE -<line number>	
Deletes every line of the current program up to and including <line number>.	`DELETE -500`

Command/Function	Examples
DELETE <line number>-<line number>	
Deletes all lines of the current program up to and including the second number.	`DELETE 10–1000`
EDIT <line number>	
Enter Edit Mode at the specified line number.	`EDIT 100`
FILES "<filename>"	
List names of files residing on the current disk.	`FILES "*.BAS"`
LIST List the program currently in memory starting with the lowest numbered line.	`LIST`
LIST <line number>	
List the specified line number.	`LIST 100`
LIST <line number>-<line number>	
List all lines from the first line up to and including the second.	`LIST 10–100`
LLIST	

List all or part of the program currently in memory. The listing will be printed on the line printer. The options for the LLIST command are the same as for the LIST command.	`LLIST`
	`LLIST 500`
	`LLIST 150–`
	`LLIST –100`
	`LLIST 150 — 400`

Command/Function	Examples
LOAD <"filename">,R	

Load a program file from disk into memory. The R is optional, and if used will run the program after it is loaded.	`LOAD"B:GAME"`
	`LOAD"PROG.ASC",R`

MERGE <"filename">	
Merges a disk file into a program in memory.	`MERGE"B:TEST.BAS"`
NEW	
Deletes the current program and clears all variables.	`NEW`
RENUM <nn>,<mm>,<ii>	

Renumbers program lines starting at line <mm>, as line <nn>, with increments of <ii>.	`RENUM`
	`RENUM 300,,5`
	`RENUM 1000,900,20`

RESET	
Changes disk in default drive.	`RESET`
RUN <line number>	

Executes the current program starting with specified line number. If line number is not specified, execution starts at the lowest line number.	`RUN 100`
	`RUN`

RUN <"filename">,R	

Loads a program from disk and executes it. R keeps all data files open.	`RUN "PROG1"`
	`RUN"B:GAME",R`

Command/Function	Examples
SAVE "filename",A SAVE "filename",P	

Saves the current program on disk. If A is used, the file is saved in ASCII format. If P is used, the file is saved in a protected format. If neither the P or A is used, the file is saved in a compressed binary format.	`SAVE"COM2",A`
	`SAVE"TEST1"`
	`SAVE"INVEN",P`

SYSTEM	
Closes all files and performs a CP/M warm start.	`SYSTEM`

Edit Mode Subcommands and Functions

Command	Function
RETURN	End editing and return to Command Mode.
<i>Space Bar	Move cursor <i> spaces to the right.
<i>Back Space	Move cursor <i> spaces to the left.
L	List remainder of program line and return cursor to the beginning of the program line.
X	List remainder of program line, move cursor to the end of the line, and go into Insert Mode.
I	Insert text beginning at the current position of the cursor. Use ESC to exit Insert Mode.
A	Cancel editing changes and return cursor to beginning of line.
E	End editing, save all changes and return to Command Mode.
Q	End editing, cancel all changes and return to Command Mode.
H	Delete remainder of line and then enter Insert Mode.
<i>D	Delete specified number of characters <i> beginning at current cursor position.
<i>C	Change (or replace) the specified number of characters <i> using the next <i> characters entered.
<i>S<c>	Move the cursor to the <i>th occurence of character <c>, counting from the current cursor position.
<i>K<c>	Delete all characters from the current cursor position up to the <i>th occurance of character <c>.

Print Using Format Field Specifiers

Numeric Specifier	Function	Example
#	Numeric field.	###
.	Decimal point position.	##.##
+	Print leading or trailing signs (plus for positive numbers, minus for negative numbers).	+##.##
–	Print trailing sign only if value printed is negative.	##.##–
**	Fill leading blanks with asterisks.	**##.##
$$	Place dollar sign immediately to left of leading digit.	$$###.##
**$	Asterisk fill and floating dollar sign.	**$###.##
,	Use comma every three digits (left of decimal point only).	##,###.##
^^^^	Exponential format. Number is aligned so leading digit is non-zero.	#.##^^^^

String Specifier	Function	Example
!	Single character	!
\<spaces>\	2+ number of spaces in character field.	\ \
&	Variable length string field.	&

Literal Specifier	Function	Example
_	Literal character string field.	

Program Statements

Statement/Function	Examples
DATA TYPE DEFINITION	
DEFINT <letter range>	
Declare range of variable names as integer data types.	`DEFINT I-N`
DEFSNG <letter range>	
Declare range of variable names as single-precision data types.	`DEFSNG A-H,O-P`
DEFDBL <letter range>	
Declare range of variable names as double-precision data types.	`DEFDBL X,Y,Z`
DEFSTR <letter range>	
Declare range of variable names as string variables.	`DEFSTR A-C,Z`
ASSIGNMENT AND ALLOCATION	
DIM <list of subscripted variables>	
Allocate storage for array.	`DIM A(20),B(12,2)`
OPTION BASE n	
Declare minimum value for array subscript. The default base is 0. This may be changed to 1.	`OPTION BASE 1`

Statement/Function	Examples
ERASE <list of array names>	
Remove an array from the program.	`ERASE A,B`
LET <variable> = <expression>	
Assign value of expression to variable.	`LET SUM = A+B+C`
REM <remark>	
Insert remark into program.	`REM GRP IS GROSS PAY`
SWAP <variable>,<variable>	
Exchange the values of two variables.	`SWAP A,B`

SEQUENCE OF EXECUTION

Statement/Function	Examples
END	
Terminate program execution, close all files and return to Command Mode.	`100 END`
FOR <V>=<X> TO <Y> STEP <Z>	
Allows repetive execution of a series of statements.	`FOR I = 1 TO 100`
GOSUB <line number>	
Branch to subroutine beginning at <line number>.	`GOSUB 100`
GOTO <line number>	
Branch to specified line number.	`GOTO 400`
NEXT <variable>	
Terminates a FOR loop.	`NEXT I`

Statement/Function	Examples
ON <expression> GOTO line1,...linek	
Evaluate expression. If INT(<expression>) equals one of the numbers 1-k, branch to appropriate line number. If it is not equal, go to the next statement.	`ON L1 GOTO 10,20,30`
ON <expression> GOSUB line1,...linek	
Same as ON...GOTO except branch is to a subroutine.	`ON L GOSUB 300,400`
RETURN	
Terminates a subroutine. Branches to the statement following the most recent GOSUB.	`RETURN`
STOP	
Terminates program execution and returns to Command Mode.	`STOP`

CONDITIONAL EXECUTION

Statement/Function	Examples
IF <expression> THEN <statement(s)> > ELSE <statement(s)>	
Evaluate <expression>: If true, execute THEN clause. If false, execute ELSE clause. (if present)	`IF A=0 THEN A=1` `ELSE A=0`

Statement/Function	Examples
WHILE <expression> . <loop statements> . WEND	

Executes a series of statements in a loop as long as a given condition is true.	`WHILE A=0`
	`PRINT "ZERO"`

NON-DISK I/O STATEMENTS	
INPUT <;> <"prompt string">;<list of variables>	
Inputs data from the terminal during program execution.	`INPUT "AGE";A`
LINE INPUT <;> <"prompt string">;<string variable>	
Inputs an entire line (up to 255 characters) to a string variable, without the use of delimiters.	`LINE INPUT J$`
DATA <list of constants>	

Stores numeric and string constants. These constants are assigned to variables by using the READ statement.	`DATA 34,23.1,45.0`
	`DATA "HELLO","BYE"`

PRINT <list of expressions>	

Outputs data on the terminal.	`PRINT "HELLO"`
	`PRINT A$,Z,C`

READ <list of variables>	

Reads data into specified variables from a DATA statement.	`READ I,A,B`
	`READ A$,B$`

Statement/Function	Examples
RESTORE <line number>	
Resets DATA pointer so that data may be reread.	`RESTORE`
LPRINT <list of expressions>	
Prints data on the line printer.	`LPRINT "HELLO"`

String Functions

Function	Operation	Example

ASC(X$)	Returns ASCII code of first character in string argument.	`ASC("B")`
		`ASC(H$)`

CHR$(I)	Returns a one-character string whose character has the ASCII code of I.	`CHR$(66)`
		`CHR$(N)`

HEX$(X)	Converts a number to a Hexadecimal string.	`HEX$(100)`
		`HEX$(A)`

INKEY$	Reads one character from the keyboard.	`A$=INKEY$`
INPUT$(X,Y)	Reads X characters from the keyboard or from file number Y.	`INPUT$(1,1)`
INSTR(I,X$,Y$)	Returns the position of the first occurrence of Y$ in X$ starting at position I.	`INSTR(A$,",")`

LEFT$(X$,I)	Returns left-most I characters of the string expression X$.	`LEFT$(A$,1)`
		`LEFT$(C$,3)`

LEN(X$)	Returns length of string X$.	`LEN(A$)`
MID$(X$,I,J)	Returns string of length J characters from X$ beginning with the Ith character.	`MID$(X$,5,10)`

Function	Operation	Example
MID$(X$,I,J)=Y$	Replaces the characters in X$, beginning at position I, with the characters in Y$. J is the number of characters to use in the replacement.	`MID$(A$,1,2)="Z"`
OCT$(X)	Conerts the numeric expression X to an octal string.	`OCT$(24)`
RIGHT$(X$,I)	Returns the right-most I characters of string X$.	`RIGHT$(X$,8)`
SPACE$(X)	Returns a string of X spaces.	`SPACE$(20)`
STR$(X)	Converts a numeric expression to a string.	`STR$(100)`
STRING$(I,J)	Returns a string of length I containing characters with the ASCII code J.	`STRING$(20,33)`
STRING$(I,X$)	Returns a string of length I containing the first character of string X$.	`STRING$(20,"!")`
VAL(X$)	Converts the string X$ to a numeric value.	`VAL("3.14")`

Arithmetic Functions

Function	Operation	Example
ABS(X)	Returns absolute value.	`ABS(-1)`
ATN(X)	Returns arctangent of X. (X must be in radians.)	`ATN(3)`
CDBL(X)	Converts X to double-precision.	`CDBL(A)`
CINT(X)	Converts X to an integer by rounding.	`CINT(46.6)`
COS(X)	Returns the cosine of X. (X must be in radians)	`COS(A+B)`
CSNG(X)	Converts X to single-precision.	`CSNG(V)`
EXP(X)	Returns e to the power of X.	`EXP(34.5)`
FIX(X)	Returns truncated integer portion of X.	`FIX(23.2)`
INT(X)	Returns largest integer not greater than X.	`INT(-12.11)`
LOG(X)	Returns the natural logarithm of X. X must be greater than zero.	`LOG(45/7)`
RND(X)	Returns a random number between 0 and 1.	`RND(0)`
SGN(X)	Returns -1 for negative X, 0 for zero X, +1 for positive X.	`SGN(C/A)`
SIN(X)	Returns the sine of X. (X must be in radians.)	`SIN(A*1.3)`
SQR(X)	Returns the square root of X. X must be non-negative.	`SQR(A*B)`
TAN(X)	Returns the tangent of X. (X must be in radians.)	`TAN(X+Y+Z)`

Special Functions

Function	Operation	Example
FRE(X)	Returns memory space not used by BASIC-80.	`FRE(0)`
INP(I)	Returns the byte read from port I.	`INP(255)`
LPOS(X)	Returns current position of line printer print head within the line printer buffer.	`LPOS(0)`
NULL(X)	Sets the number of nulls to be printed at the end of each line.	`NULL(3)`
OUT I,J	Sends byte J to port I.	`OUT 127,255`
PEEK(I)	Reads a byte from the specified memory address.	`PEEK(8192)`
POKE I,J	Puts byte J into memory location I.	`POKE(8192,200)`
POS(X)	Returns current cursor position.	`POS(1)`
SPC(I)	Prints I spaces on the terminal.	`PRINT SPC(5)`
TAB(I)	Tabs carriage to specified position.	`PRINT TAB(20)`
VARPT(X)	Returns address of variable in memory.	`VARPTR(V)`
WAIT I,J[,K]	Status of port I is XOR'ed with K and AND'ed with J. Continued execution awaits non zero result.	`WAIT 21,1`
WIDTH I	Sets the printed line width.	`WIDTH 80`
WIDTH LPRINT I	Sets the line printer width.	`WIDTH LPRINT 132`

Special Features

ERROR TRAPPING

Statement/Function	Example
ON ERROR GOTO <line number>	
Enables error trapping and specifies the first line of the error trapping subroutine.	`ON ERROR GOTO 100`
RESUME <line number>	

Continues program execution after an error recovery procedure has been performed.	`RESUME`
	`RESUME NEXT`
	`RESUME 100`

ERROR <integer expression>	
Simulates the occurance of an error, also allows error codes to be defined by user.	`ERROR 10`
ERL Error line number.	`PRINT ERL`
ERR Error code number.	`PRINT ERR`

TRACE FLAG

Statement/Function	Example
TRON Enables trace flag.	`TRON`
TROFF Disables trace flag.	`TROFF`

Statement/Function	Example
OVERLAY MANAGEMENT	
CHAIN [MERGE]"<filename>"[,[<line number>] [,ALL][,DELETE<range>]]	
Calls program and passes variables from the current program.	`CALL "PROG"`
COMMON <list of variables>	
Pass variables to a chained program.	`COMMON A,B`

Disk Input/Output Statements

Statement/Function	Example
CLOSE#[<filenumber>[,<filenumber>]	
Closes disk files. If no argument is supplied, all open files are closed.	`CLOSE #6`
FIELD# <filenumber>,<field size> AS <string variable>	
Allocates random buffer space to <string variable>, where <file number> is the random buffer referenced, and <field size> is the space reserved for a given <string variable>.	`FIELD #1,3 AS A$`
GET#<file number>[,<record number>]	
Transfers data from the <record number> of the random file <file number> to the random buffer. If <record number> is omitted, the next record is transferred.	`GET #1,I`
INPUT#<filenumber>,<variable list>	
Reads data from file <filenumber> and assigns the input to the elements of <variable list>.	`INPUT #3,A,B`
KILL "<filename>"	
Deletes a disk file.	`KILL "A:GAME.BAS"`
LINE INPUT#<file number>,<string variable>	
Read an entire line from a file <file number> and assigns it to <string variable>.	`LINE INPUT #1,A$`

Statement/Function	Example
LSET <string variable> = <string expression>	
Stores data in random file buffer, left justified.	`LSET A$="HELLO"`
OPEN <mode>,[#]<filenumber>,<"filename">	
Opens a disk file, where <mode> is the file type,<filenumber> is the I/O label, and <file name> is the disk directory entry.	`OPEN "O",1,"GM.DAT"`
PRINT#<file number>,<list of expressions>	
Writes data to a sequential disk file.	`PRINT #1,A$,B`
PUT [#]<filenumber>[,<record number>]	
Transfers data from the random file buffer to random file <file number>. If <record number> is omitted, the next record is written.	`PUT #2,3`
RSET <string variable> = <string expression>	
Stores data in a random file buffer, right justified.	`RSET B$="BYE"`
WRITE#<file number>,<list of expressions>	
Writes data to a sequential disk file. Delimiters are inserted between items in the I/O list.	`WRITE #2,A,B$`

Disk Input/Output Functions

Function	Operation	Example
CVD(X$)	Converts 8-character string to double precision number.	`A#=CVD(A$)`
CVI(X$)	Converts 2-character string to an integer.	`I%=CVI(I$)`
CVS(X$)	Converts 4-character string to single precision number.	`B=CVS(B$)`
EOF(file no.)	Returns true (−1) if a file is positioned at its end.	`IF EOF(1)`
LOC(file no.)	Returns next record number to read (random file). Returns number of sectors accessed (sequential file).	`X=LOC(1)`
MKD$(Z#)	Converts double-precision number to an 8-character string.	`A$=MKD$(A#)`
MKI$(I%)	Converts an integer to a 2-character string.	`I$=MKI$(I%)`
MKS$(B)	Converts a single-precision number to a 4-character string.	`B$=MKS$(B)`

INSERT

Appendix A

Error Messages

After an error occurs, BASIC-80 returns to the Command Mode and types Ok. (Although overflow and division by zero errors will not cause BASIC-80 to stop execution.) Variable values and the program text remain intact, but you cannot continue the program with the CONT command. However, execution can be continued with a Command Mode GOTO.

The formats of error messages are:

Direct Statement	<error message>
Indirect Statement	<errror message> in nnnnn

where nnnnn is the line number where the error occurred. When an error occurs in a direct statement, no line number is printed.

The error messages are listed on the next few pages, along with the error number. If an error should occur for which there is no error code, BASIC-80 will print the message "Unprintable error".

GENERAL ERRORS

1 `NEXT without FOR`

The variable in a NEXT statement corresponds to no previously executed FOR statement.

2 `Syntax error`

A line has been encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled statement or command, incorrect puncuation, etc.).

3 `RETURN without GOSUB`

A RETURN statement has been encountered before a GOSUB was executed.

4 `Out of data`

A READ statement was executed but all of the DATA statements in the program have already been read.

5 `Illegal function call`

The parameter passed to an arithmetic or string function was out of range. Illegal function calls can occur due to:

1. A negative array subscript (LET A(-1)=0).
2. An unreasonably large array subscript (>32767).
3. LOG with a negative or zero argument.
4. SQR with a negative argument.
5. A^B with A negative and B not an integer.
6. A call to a USR function before the address of a machine language subroutine has been entered.
7. Calls to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO with an improper argument.

6 Overflow

The result of a calculation was too large to be represented in BASIC-80's number format. If an underflow (i.e. a number is too small to be represented) occurs, zero is given as the result and execution continues without any error message being printed.

7 Out of memory

A program is too large, has too many variables, too many FOR loops, too many GOSUB's, or too complicated expressions.

8 Undefined line number

The line reference in a GOTO, GOSUB, IF...THEN...ELSE or DELETE was to a non-existent line.

9 Subscript out of range

An attempt was made to reference an array element which is either outside the dimensions of the array, or with the wrong number of subscripts.

10 Duplicate Definition

After an array was dimensioned, another dimension statement for the same array was encountered. The error often occurs if an array was given the default dimension of 10 and later in the program the same array is specified in a DIM statement.

11 Division by zero

A division by zero has been encountered in an expression, or the evaluation of an expression results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.

12 Illegal direct

A statement that is illegal in Direct Mode has been entered as a Direct Mode command.

```
13        Type mismatch
```

A string variable has been assigned a numeric value or vice versa; a function that expects a numeric argument has been given a string argument or vice versa.

```
14        Out of string space
```

String variables have caused BASIC-80 to exceed the amount of free memory remaining. BASIC-80 will allocate string space dynamically, until it runs out of memory.

```
15      String too long
```

An attempt was made to create a string more than 255 characters long.

```
16        String formula too complex
```

A string expression was too long or too complex. The expression should be broken into smaller expressions.

```
17        Can't continue
```

An attempt has been made to continue a program that:

1. Has halted due to an error.
2. Has been modified during a break in execution.
3. Does not exist.

```
18        Undefined user function
```

A reference was made to a user-defined function which had never been defined.

```
19        No RESUME
```

BASIC-80 entered an error trapping routine, but the program ended before a RESUME statement was encountered.

```
20        RESUME without error
```

A RESUME statement was encountered, but no error trapping routine had been entered.

`21        Unprintable error`

An error message is not available for the error condition which exists. This is usually caused by an ERROR with an undefined error code.

`22        Missing operand`

During evaluation of an expression, an operator was found with no operand following it.

`23        Line buffer overflow`

An attempt has been made to input a line that has too many characters.

`26        FOR without NEXT`

A FOR was encountered without a matching NEXT.

`29        WHILE without WEND`

A WHILE statement has been encountered without a matching wend.

`30        WEND without WHILE`

A WEND was encountered without a matching WHILE.

DISK RELATED ERRORS

`50        Field overflow`

An attempt was made to allocate more bytes than were specified for the record length of a random file.

`51        Internal error`

An internal malfunction has occurred in BASIC-80. Report conditions under which error occurred and all relevant data to Zenith Data Systems Customer Service.

`52        Bad file number`

A statement or command has referenced a file number that is not OPEN or is out of the range of numbers specified at initialization.

`53        File not found`

A LOAD, KILL, or OPEN statement referenced a file that did not exist.

`54        Bad file mode`

An attempt was made to perform a PRINT or WRITE on a random file, to OPEN an already open random file for sequential output, to perform a GET or PUT on a sequential file, to load from a random file, or to execute an OPEN statement where the file mode is not I,O, or R.

`55        File already open`

A sequential output mode is issued for a file that is already open; or a KILL is given for a file that is open.

`57        Disk I/O error`

An I/O error occured on a disk I/O operation. It is a fatal error, i.e., the operating system cannot recover from the error.

`58        File already exists`

The file name specified in a NAME statement is identical to a file name already in use on the disk.

`61        Disk full`

All disk storage space is in use.

`62        Input past end`

An INPUT statement is executed after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end of file.

`63        Bad record number`

In a PUT or GET statement, the record number is either greater than the maximum allowed (32768) or equal to zero.

`64        Bad file name`

An illegal form is used for the file name with LOAD, SAVE, KILL, or OPEN.

`66        Direct statement in file`

A direct statement is encountered while an ASCII-format file is being loaded. The LOAD is terminated.

`67        Too many files`

An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.

RESERVED WORDS

Some words are reserved by BASIC-80 for use as statements, commands, operators, and so on, and therefore may not be used in variable or function names. The reserved words are listed below. Note that all intrinsic functions are considered to be reserved.

ABS	AND	ASC	ATN
AUTO	BASE	CALL	CHAIN
CINT	CDBL	CHR$	CLEAR
CLOSE	COMMON	CONT	COS
CSNG	CVD	CVI	CVS
DATA	DEF	DEFDBL	DEFINT
DEFSNG	DEFSTR	DEFUSR	DELETE
DIM	EDIT	ELSE	END
EOF	ERASE	ERL	ERR
ERROR	EXP	FIELD	FILES
FIX	FN	FOR	FRE
GET	GOSUB	GOTO	HEX$
IF	IMP	INKEY$	INP
INPUT	INSTR	INT	KILL
LEFT$	LEN	LET	LINE
LIST	LLIST	LOAD	LOC
LOF	LOG	LPOS	LPRINT
LSET	MERGE	MID$	MKD$
MKI$	MKS$	MOD	NAME
NEW	NEXT	NOT	NULL
OCT$	ON	OPEN	OPTION
OR	OUT	PEEK	POKE
POS	PRINT	PUT	RANDOMIZE
READ	REM	RENUM	RESET
RESTORE	RESUME	RETURN	RIGHT$
RND	RSET	RUN	SAVE
SGN	SIN	SPACE$	SPC
SQR	STEP	STOP	STR$
STRING$	SWAP	SYSTEM	TAB
TAN	THEN	TO	TROFF
TRON	USR	VAL	VARPTR
WAIT	WEND	WHILE	WIDTH
WRITE	XOR		

Appendix B

ASCII Codes

DECIMAL TO OCTAL HEX TO ASCII CONVERSION

I				II				III				IV			
DEC	OCT	HEX	ASCII	DEC	OCT	HEX	ASCII	DEC	OCT	HEX	ASCII	DEC	OCT	HEX	ASCII
0	000	00	NUL	32	040	20	SPACE	64	100	40	@	96	140	60	'
1	001	01	SOH	33	041	21	!	65	101	41	A	97	141	61	a
2	002	02	STX	34	042	22	"	66	102	42	B	98	142	62	b
3	003	03	ETX	35	043	23	#	67	103	43	C	99	143	63	c
4	004	04	EOT	36	044	24	$	68	104	44	D	100	144	64	d
5	005	05	ENQ	37	045	25	%	69	105	45	E	101	145	65	e
6	006	06	ACK	38	046	26	&	70	106	46	F	102	146	66	f
7	007	07	BEL	39	047	27	'	71	107	47	G	103	147	67	g
8	010	08	BS	40	050	28	(	72	110	48	H	104	150	68	h
9	011	09	HT	41	051	29	)	73	111	49	I	105	151	69	i
10	012	0A	LF	42	052	2A	*	74	112	4A	J	106	152	6A	j
11	013	0B	VT	43	053	2B	+	75	113	4B	K	107	153	6B	k
12	014	0C	FF	44	054	2C	,	76	114	4C	L	108	154	6C	l
13	015	0D	CR	45	055	2D	–	77	115	4D	M	109	155	6D	m
14	016	0E	SO	46	056	2E	PERIOD	78	116	4E	N	110	156	6E	n
15	017	0F	SI	47	057	2F	/	79	117	4F	O	111	157	6F	o
16	020	10	DLE	48	060	30	0	80	120	50	P	112	160	70	p
17	021	11	DC1	49	061	31	1	81	121	51	Q	113	161	71	q
18	022	12	DC2	50	062	32	2	82	122	52	R	114	162	72	r
19	023	13	DC3	51	063	33	3	83	123	53	S	115	163	73	s
20	024	14	DC4	52	064	34	4	84	124	54	T	116	164	74	t
21	025	15	NAK	53	065	35	5	85	125	55	U	117	165	75	u
22	026	16	SYN	54	066	36	6	86	126	56	V	118	166	76	v
23	027	17	ETB	55	067	37	7	87	127	57	W	119	167	77	w
24	030	18	CAN	56	070	38	8	88	130	58	X	120	170	78	x
25	031	19	EM	57	071	39	9	89	131	59	Y	121	171	79	y
26	032	1A	SUB	58	072	3A	:	90	132	5A	Z	122	172	7A	z
27	033	1B	ESC	59	073	3B	;	91	133	5B	[	123	173	7B	{
28	034	1C	FS	60	074	3C	<	92	134	5C	\	124	174	7C	|
29	035	1D	GS	61	075	3D	=	93	135	5D	]	125	175	7D	}
30	036	1E	RS	62	076	3E	>	94	136	5E	△	126	176	7E	~
31	037	1F	US	63	077	3F	?	95	137	5F	_	127	177	7F	DELETE

Control Character Definitions

NUL	Null: Tape feed,
SOH	Start of Heading; Start of Message
STX	Start of Text; End of Address
ETX	End of Text; End of Message
EOT	End of Transmission; Shuts off TWX machines
ENQ	Enquiry; WRU
ACK	Acknowledge; RU
BEL	Rings Bell
BS	Backspace
HT	Horizontal TAB
LF	Line Feed or Space (New Line)
VT	Vertical TAB
FF	Form Feed (PAGE)
CR	Carriage Return
SO	Shift Out
SI	Shift In
DLE	Data Link Escape
DC1	Device Control 1; Reader on
DC2	Device Control 2; Punch on
DC3	Device Control 3; Reader off
DC4	Device Control 4; Punch off
NAK	Negative Acknowledge; Error
SYN	Synchronous Idle(SYNC)
ETB	End of Transmission Block; Logical End of Medium
CAN	Cancel (CANCL)
EM	End of Medium
SUB	Substitute
ESC	Escape
FS	File Separator
GS	Group Separator
RS	Record Separator
US	Unit Separator

Refer to the chart on Page B-1. Note that any print control character defined above and listed in column I of the chart can be produced from the combination of CTRL and the alphabetical character in column III or IV which is on the same line and to the right of the print control character. That is, DLE is CTRL-P or ^P, BEL is CTRL-G or ^G, and so on.

Appendix C

New Features in BASIC-80

New Reserved Words

BASIC-80 has new reserved words: CALL, CHAIN, COMMON, WHILE, WEND, WRITE,OPTION BASE, RANDOMIZE.

Type Conversions

Conversion from floating point to integer values results in rounding. (Previous versions of Microsoft BASIC would truncate the value.) This affects not only assignment statements (e.g., I%=2.5 results in I%=3), but also affects function and statement evaluations [e.g., TAB(4.5) goes to the fifth position, A(1.5) yields A(2), and X=11.5 MOD 4 yields 0]

FOR/NEXT Loop Evaluation

The body of FOR/NEXT loop is skipped if the initial value of the loop exceeds the terminal value (or if a negative STEP is specified and the initial value is less than the terminal value). See Chapter Four, "Program Statements," for more information about FOR/NEXT loops.

Division by Zero and Overflow

Division by zero and overflow no longer produce fatal errors. See Chapter Two, "Expressions," for more information.

RND Function

The RND function has been changed so that RND with no argument is the same as RND with a positive argument. The RND function generates the same sequence of random numbers each time it is executed. The RANDOMIZE option should be used to reseed the random number generator. See Chapter Seven, "Functions," for more information.

Printing Numeric Values

The rules for PRINTing single-precision and double-precision numbers have been changed. See Chapter Four, "Program Statements," for more information about the PRINT statement.

String Space Allocation

String space is allocated dynamically, so the CLEAR statement is no longer used to set aside memory for string storage. The first argument in a CLEAR statement is used to set the end of memory, and the second argument is used to set the amount of stack space.

Invalid Input

Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.), or with only a carriage return causes the message "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

PRINT USING Characters

There are two new field formatting characters for use with PRINT USING. An ampersand is used for variable length string fields, and an underscore signifies a literal character in a format string.

WIDTH Statement

If the expression supplied with the WIDTH statement is 255, BASIC-80 uses an "infinite" line width; that is, it does not insert carriage returns. WIDTH LPRINT may be used to set the line width at the line printer.

EDIT Characters

The at-sign and underscore are no longer used as editing characters.

Variable Names

Variable names are significant up to 40 characters and can contain embedded reserved words. However, reserved words must now be delimited by spaces. To maintain compatibility with earlier versions of BASIC, spaces will be automatically inserted between adjoining reserved words and variable names. This insertion of spaces may cause the end of a line to be truncated if the line length exceeds 255 characters.

Protected Binary Format

BASIC-80 programs may be saved in a protected binary format so that they may not be LISTed or EDITed.

Appendix D

Programming Hints

As your level of programming experience increases, you will eventually have to concern yourself with program efficiency. The two main resources you will have to conserve are: memory space and execution time. This Appendix has been included to aid in your programming effort.

CONSERVING MEMORY SPACE

To conserve memory space, make sure that you do the following:

Place multiple program statements on a single line.

BASIC-80 must keep track of each program line as well as the program line number. If you place multiple statements on a single line, less space will be used for program line overhead.

Remove all unnecessary REM statements.

When you use a REM statement, BASIC-80 will store the one-byte code which represents the REM keyword plus the ASCII representation of the actual remark. This can result in a lot of memory being used simply for remarks. (You will have to consider the trade-off of program documentation vs. memory space when you remove these REM statements.)

Use a subroutine call (GOSUB) only when a GOTO won't work.

The GOSUB statement should be used only when a routine must be called from several different places within the main program. If a routine is to be called from the same place every time, then use a GOTO. Each active GOSUB will consume memory space (to update the stack), but a GOTO will not.

Use as few parentheses in an expression as possible.

Structure your arithmetic expressions so they use as few parentheses as possible. Each time BASIC-80 has to evaluate an expression enclosed in parentheses, it will consume more memory space. BASIC-80 will also have to store the result of this evaluation in a temporary storage location, thus using more memory space.

Use integer variables whenever possible.

This is very important, as integer variables only consume two bytes of memory. A single-precision variable will take four bytes, and a double-precision will take eight bytes.

Dimension arrays sparingly.

Make sure that you only allocate as much space for an array as you will use. For example, if you allow BASIC-80 to establish the 11-element default array size, and then only use four of these elements, you have wasted more space than you have used. So always set the array size with a dimension statement, never let BASIC-80 assume the default size of 11 elements. (Unless your array size is only 11 elements.)

Split large programs into smaller modules.

BASIC-80 will allow you to CHAIN between programs, as well as pass variables between programs. This makes it very easy to write a large program as several small programs and pass variables between them.

Use DEF statements to declare variable types.

This will prevent you from having to use the type declaration characters, thus saving you one byte for every variable that is not a single-precision data type.

Reduce the number of simultaneously open data files.

Every data file requires a buffer area, so it is more efficient to use the same buffer for several different files. To do this, open the first file as file #1, and then access it as needed. Then close this file and open the second file as file #1. Although you will not be able to simultaneously access both files, you will still be able to access both files as needed.

Reduce the number of variables and arrays in a program.

You can accomplish this by reusing variables and arrays in a program when they are no longer needed. Or, you can establish one variable to be used as a FOR/NEXT counter, and then use it for every FOR/NEXT loop.

SAVING EXECUTION TIME

To save execution time make sure you do the following:

Define the most commonly used variables first.

The variables are placed in the BASIC-80 variable table as they are encountered. When a variable is referenced, the table is searched sequentially. Thus, if a variable is near the top of the table, it will take less time to access.

Use integer variables in FOR/NEXT loops.

This is very important and can result in a significant time savings. If you wish to try an experiment, set up a FOR/NEXT with a single-precision loop counter and time the execution. Then simply define the loop counter as an integer data type and time the execution again. (Make sure you set the loop for at least 10,000 iterations.) You will notice a significant difference in the execution times.

Use variables instead of constants in arithmetic expressions.

BASIC-80 uses a floating point decimal representation for numeric values. It takes less time for BASIC-80 to access a variable than to convert a constant to this representation. If you have a constant you are planning to use quite often in a program, assign it to a variable and use the variable instead.

This list is by no means exhaustive, but if you adhere to the above suggestions, you will be well on the way to generating efficient code.

Appendix E

Assembly Language Subroutines

BASIC-80 provides two methods for calling assembly language subroutines from a BASIC-80 program. The first method uses the USR function, which allows assembly language subroutines to be called in the same way BASIC-80's intrinsic functions are called. The second method uses the CALL statement, which generates the same calling sequence as the Microsoft FORTRAN, COBOL, and BASIC Compilers.

Since assembly language subroutines bypass some of the built-in safeguards of BASIC-80, calling assembly language subroutines renders BASIC-80 vulnerable to and defenseless against the errors in those subroutines. Therefore, write your subroutines with caution.

MEMORY ALLOCATION

When using assembly language subroutines with BASIC-80, an important consideration is memory space allocation. Memory space must be set aside for an assembly language subroutine before it can be loaded.

During initialization, enter the highest memory location minus the amount of memory needed for the assembly language subroutine(s). The /M switch can be used during initialization to set the top of memory. (See Chapter One, "System Introduction & General Information," for more information about the initialization procedure.) BASIC-80 uses all memory available from its starting location up, so only the topmost locations in memory can be set aside for user subroutines.

After an assembly language subroutine is called, the stack pointer is set up for eight levels (16 bytes) of stack storage. If more stack space is needed, BASIC-80's stack can be saved and a new stack set up for use by the assembly language subroutine. BASIC-80's stack must be restored, however, before the program returns from the subroutine.

The assembly language subroutine may be loaded into memory by means of the CP/M system monitor, or by using the BASIC-80 POKE statement. Assembly language subroutines may also be assembled with the MACRO-80 assembler and loaded using the LINK-80 linking loader. (These programs are not provided with BASIC-80, they must be purchased separately.)

USR FUNCTION CALLS

Before a USR function is called, the entry address for the USR subroutine must be defined in a DEF USR statement.

DEF USR
(define entry address for USR subroutine)

Form: DEF USR<digit>=<expression>

The DEF USR statement is used to define entry points for up to 10 assembly language subroutines.

The <digit> is the number of the assembly language subroutine. <digit> may be any number from 0-9. If <digit> is omitted, it it assumed to be 0.

The value of <expression> is the starting address of the assembly language subroutine. This address is assumed to be in decimal unless a special base specifier character is used. Hexadecimal numbers are specified with the prefix &H and octal numbers are specified with the prefix &O or &.

The format of the USR function call is:

USR[<digit>](argument)

where <digit> is from 0 to 9 and the argument is any numeric or string expression. <digit> specifies which USR subroutine is being called, and corresponds with the digit supplied in the DEF USR statement for that subroutine. If <digit> is omitted, USR0 is assumed. The address given in the DEF USR statement determines the starting address of the subroutine.

When the USR function call is made, register A contains a value that specifies the data type of the argument that was given. The value in A will be one of the following:

Value in A	Type of Argument
2	Two-byte integer (two's complement)
3	String
4	Single-precision floating point number
8	Double-precision floating point number

Table E-1
Register Values Used to Specify Data Types.

If the argument is a numeric data type, the [H,L] register pair will point to the Floating Point Accumulator (FAC) where the argument is stored. The FAC occupies eight bytes in memory — enough for a double-precision number.

NUMERIC STORAGE FORMAT

Integer Storage Format

An integer argument is stored as a 2-byte data value. The integer is stored in a two's complement representation.(In the following discussion, the Floating Point Accumulator will be referred to as the FAC.) An integer argument will be stored in the FAC as follows:

FAC-3 — Contains the lower 8 bits of the argument
(the least significant byte)

FAC-2 — Contains the upper 8 bits of the argument
(the most significant byte)

Single-Precision Storage Format

A single-precision argument is stored as a 4-byte data value. The first byte will be the exponent. The exponent will be stored in excess 128 (200 octal) notation. This means that 200 (octal) represents an exponent of 0, 201 (octal) represents an exponent of 1, 177 (octal) represents an exponent of -1, and so forth. A single-precision number will be stored in the FAC as follows:

FAC-3 — Contains the lowest eight bits of the mantissa.

FAC-2 — Contains the middle eight bits of the mantissa.

FAC-1 — Contains the highest seven bits of the mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0=positive, 1=negative).

FAC — Contains the exponent stored in "excess 128" (200 octal) format

Double-Precision Storage Format

A double-precision argument is stored using the same format as the single-precision number, only four more bytes are used to store the mantissa. A double-precision number is stored in the FAC in the same manner as a single-precision number, except:

FAC-7 through FAC-4 contain four more bytes of the mantissa (FAC-7 contains the lowest eight bits).
(least significant).

STRING STORAGE FORMAT

If the argument is a string, the [D,E] register pair points to three bytes called the "string descriptor". Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes one and two, respectively, are the lower and upper eight bits of the string starting address in string space.

CAUTION: If the argument is a string literal in the program, the string descriptor will point to the program text where the string appears. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program.

Example:

```
A$ = "BASIC-80"+""
```

This will force BASIC-80 to copy the string literal into string space and will prevent alteration of program text during a subroutine call.

Data Type Conversions

Usually, the value returned by a USR function is the same type (integer, string, single-precision or double-precision) as the argument that was passed to it. However, calling the MAKINT subroutine returns the integer in [H,L] as the value of the function, thus forcing the value returned by the function to be integer.

To execute MAKINT, use the following sequence to return from the subroutine:

```
MAKINT  EQU    105H      ;address of MAKINT for CP/M
        PUSH   H         ;save value to be returned
        LHLD   MAKINT    ;get address of MAKINT subroutine
        XTHL             ;save return on stack and
                         ;get back [H,L]
        RET              ;return
```

Also, the argument of the function, regardless of its type, may be forced to an integer value of the argument in [H,L]. Execute the following subroutine:

```
FRCINT  EQU    103H       ;address of FRCINT for CP/M
        LXI    H          ;get address of subroutine
                          ;continuation
        PUSH   H          ;place on stack
        LHLD   FRCINT     ;get address of FRCINT
        PCHL
```

CALL STATEMENT

BASIC-80 user function calls may also be made with the CALL statement. The calling sequence used is the same as that in Microsoft's FORTRAN, COBOL and BASIC compilers.

The general format of the CALL statement is:

```
CALL <variable name>[(argument list)]
```

<variable name> is assigned an address that is the starting point in memory of the assembly language subroutine. The address should be assigned to <variable name> before a CALL statement is executed. <variable name> may not be an array variable name. <argument list> contains the arguments that are passed to the assembly language subroutine.

A CALL statement with no arguments generates a simple "CALL" instruction. The corresponding subroutine should return via a simple "RET." (CALL and RET are 8080 opcodes - consult an 8080 reference manual for details.)

A subroutine CALL with arguments results in a somewhat more complex calling sequence. For each argument in the CALL argument list, a parameter is passed to the subroutine. That parameter is the address of the low byte of the argument. Therefore, parameters always occupy two bytes each, regardless of data type.

The method of passing the parameters depends upon the number of parameters to pass:

A. If the number of parameters is less than or equal to 3, they are passed in the registers. Parameter 1 will be in HL, 2 in DE (if present), and 3 in BC (if present).

B. If the number of parameters is greater than 3, they are passed as follows:

   1. Parameter 1 in HL.

   2. Parameter 2 in DE.

   3. Parameters 3 through n in a contiguous data block. BC will point to the low byte of this data block (i.e., to the low byte of parameter 3).

Note that, with this scheme, the subroutine must know how many parameters to expect in order to find them.

Conversely, the calling program is responsible for passing the correct number of parameters. There are no checks for correct number or type of parameters.

If a subroutine expects more than three parameters, and needs to transfer them to a local data area, there is a system subroutine named $AT (located in the FORTRAN library, FORLIB.REL) which will perform the transfer. If you do not have FORTRAN, the $AT argument transfer subroutine is listed on Page E-9.

$AT is called with HL pointing to the local data area, BC pointing to the third parameter, and A containing the number of arguments to be transferred (i.e., the total number of arguments minus 2). Your subroutine is responsible for saving the first two parameters before calling $AT.

For example, if a subroutine expects five parameters, it should use the following general procedure:

```
SUBR:     SHLD    P1          ;SAVE PARAMETER 1
          XCHG
          SHLD    P2          ;SAVE PARAMETER 2
          MVI     A,3         ;NO. OF PARAMETERS LEFT
          LXI             ;POINTER TO LOCAL AREA
          CALL    $AT         ;TRANSFER THE OTHER 3 PARAMETERS
          .
          .
          .
          [body of subroutine]
          .
          .
          .
          RET                 ;RETURN TO CALLER
P1:       DS      2           ;SPACE FOR PARAMETER 1
P2:       DS      2           ;SPACE FOR PARAMETER 2
P3:       DS      6           ;SPACE FOR PARAMETERS 3-5
```

When parameters are accessed in a subprogram, remember that they are only pointers to the actual arguments passed.

It is entirely up to the programmer to insure that the arguments in the calling program correspond in number, type, and length with the parameters expected by the subprogram.

A listing of the argument transfer subroutine $AT follows.

```
;          ARGUMENT TRANSFER
;[B, C]     POINTS TO 3RD PARAMETER
;[H, L]     POINTS TO LOCAL STORAGE FOR PARAMETER 3
;[A]       CONTAINS THE # OF PARAMETERS TO XFER(TOTAL-2)


            ENTRY   $AT
$AT:        XCHG                    ;SAVE [H,L] IN [D,E]
            MOV     H,B
            MOV     L,C             ;[H,L] = PTR TO PARAMETERS
AT1:        MOV     C,M
            INX     H
            MOV     B,M
            INX     H               ;[B,C] = PARAM ADR
            XCHG                    ;[H,L] POINTS TO LOCAL STORAGE
            MOV     M,C
            INX     H
            MOV     M,B
            INX     H               ;STORE PARAM IN LOCAL AREA
            XCHG                    ;SINCE GOING BACK TO AT1
            DCR     A               ;TRANSFERRED ALL PARAMS?
            JNZ     AT1             ;NO, COPY MORE
            RET                     ;YES, RETURN
```

INTERRUPTS

Assembly language subroutines can be written to handle interrupts. All interrupt handling subroutines should save the stack, registers A-L, and the PSW. Interrupts should always be re-enabled before returning from the subroutine, since an interrupt automatically disables all further interrupts once it is received. It is also very important to choose the proper interrupt vector. With CP/M BASIC-80, all interrupt vectors are free.

Appendix F

Random and Sequential I/O Programming Examples

A directory application, such as a computerized telephone book, is a practical use of random files. The following two sample programs illustrate this technique. The first program, "DIRECTORY", accepts the data required to build the random file and a sequential directory file. The second program, "QUERY", retrieves the data from the directory file.

To fully understand this method of random I/O, you should look at what information is contained in the directory file. The directory file has a key created from putting together the individual's first and last names. The other field in the directory is the record number. The record number is used as an index, and points to that particular individual's entry in the random file.

When you run the "QUERY" program, you will supply the first and last name of a person. If it is a valid name (that is, if it is an entry in the directory), the record number will be used. This will point to the proper record in the random file, so the telephone number can be retrieved.

Note that these examples are NOT intended to be efficient examples of random file usage. They are designed to show how to use the random and sequential file commands.

The example does not show how to add to the data in the file once it has been created. This was done to keep the example simple. If you want to add more names to the file, you will need to modify or rewrite the build program.

As it stands, the build program assumes that there is no pre-existing directory file and starts building one. If it were changed to read in the old directory file, then new entries could be added. (Lines 50 to 80 in the query program read the file.)

If you want to do this, first open A:TABLE.EXT for input and read all of it into an array such as NP$ and SP. Then close the file, but reopen it for output before you write out the directory.

Again, this example is not designed to be efficient. An efficient program would put the directory as the first or last few records of the file A:RFILE.EXT. In addition, the directory would be kept in alphabetical order for efficient searching.

You will understand these examples best if you type them in and use them.

```
5 REM "DIRECTORY PROGRAM"
10 OPEN "O",1,"A:TABLE.EXT"
20 OPEN "R",2,"A:RFILE.EXT"
30 FIELD #2,12 AS LN$,9 AS SN$,12 AS SR$<operator types LINE FEED>
     12 AS CI$, 10 AS SZ$,2 AS CD$,2 AS EX$,2 AS PN$
40 REC=REC+1
50 LINE INPUT "LAST NAME? ":N1$
60 LINE INPUT "FIRST NAME? ":N2$
70 LINE INPUT "STREET ADDRESS? ";N3$
80 LINE INPUT "CITY?" ";N4$
90 LINE INPUT "STATE ZIP? ";N5$
100 INPUT "PHONE NUMBER (XXX,XXX,XXXX) ";N%,N1%,N2%
110 LSET LN$=N1$:LSET SN$=N2$:LSET SR$=N3$:<operator types LINE FEED>
     LSET CI$=N4$:LSET SZ$=N5$
120 LSET CD$=MKI$(N%) :LSET EX$=MKI$(N1%)<operator types LINE FEED>
     :LSET PN$=MKI$(N2%)
130 KEY$=N1$+N2$
140 PRINT #1,KEY$;",",REC
150 PUT #2,REC
160 LINE INPUT "MORE INPUT (Y OR NO)";MI$<operator types LINE FEED>
     :IF MI$="Y" GOTO 40
170 CLOSE
180 END
```

Line Number	Explanation
10	Open directory file and label it "A:TABLE.EXT".
20	Open a random file and label it as "A:RFILE.EXT."

30	Reserve space in the random file buffer for directory entries.
	
	LN$=Last Name
	SN$=First Name
	SR$=Street Address
	CI9=City
	SZ$=State and Zip Code
	CD$=Area Code
	EX$=Telephone Exchange
	PN$=Last 4 digits of telephone number

40	Increment record number counter.
50 - 100	Accept input data.
110	Left-justify the string input for the random buffer.
120	Left-justify and convert integers to string values. (You must convert to strings before PUTting values into the buffer.)
130	Construct the key from first and last names.
140	Output data to the directory file. KEY$=Key for directory REC=Record number of random file
150	Put the record in the random buffer.
160	Check for more data.
170	Close all files.
180	End the program and return to MBASIC Command Mode.

```
5 REM "QUERY PROGRAM"
10 CLEAR 200
20 OPEN "I",1,"A:TABLE.EXT"
30 OPEN "R",2,"A:RFILE.EXT"
40 FIELD #2,12 AS LN$,9 AS SN$,12 AS SR$,12 AS CI$,<operator types LINE FEED>
     10 AS SZ$,2 AS CD$,2 AS EX$,2 AS PN$
50 IF EOF(1) THEN GOTO 90
60 CT=CT+1
70 INPUT #1,NP$(CT),SP(CT)
80 GOTO 50
90 INPUT "NAME (LAST,FIST)";L$,F$
100 KEY$=L$+F$
110 FOR I%=1 TO CT
120 IF KEY$=NP$(I%) THEN GO TO 150
130 NEXT I%
140 PRINT "NO RECORD EXIST":GOTO 170
150 GET #2,SP(I%)
160 PRINT LN$,SN$,CVI(CD$);"-";CVI(EX$);"-";CVI (PN$)
170 INPUT"MORE QUERIES? (Y OR N) ";M$:IF M$="Y"GOTO 90
180 CLOSE
190 END
```

Line Number	Explanation
10	Set up string storage space.
20	Open directory file for input.
30	Open the random file.
40	Reserve space in random file buffer.
50	Check for end-of-file condition.
60	Increment directory record counter.
70	Read directory into string.
80	Loop back for EOF check.
90	Supply the name for which you want the telephone number.
100	Create key from the first and last names.
110	Set up loop to search for record in the directory.
120	Compare input key to directory key.
130	If no match on first comparison, try the next key.
140	If no match is found after comparing all keys, print the message.
150	If match is found, put the requested record in the random buffer.
160	After converting the requested record back to integer, print it.
170	Check for more queries.
180	Close all files.
190	End the program and return to Microsoft BASIC's prompt.

INSERT

Index