WRITE # Statement

Purpose: Writes data to a sequential file.

Versions:	Cassette	Disk	Advanced	Compiler
	***	***	***	***

Format: WRITE *#filenum, list of expressions*

Remarks: *filenum* is the number under which the file was opened for output.

list of expressions
is a list of string and/or numeric expressions, separated by commas or semicolons.

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

WRITE # Statement

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

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

writes the following image to the file.

"CAMERA","93604-1"

A subsequent INPUT # statement, such as:

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

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

APPENDIXES

Contents

NOTES

Appendix A. Messages

If BASIC detects an error that causes a program to stop running, an error message is displayed. It is possible to trap and test errors in a BASIC program using the ON ERROR statement and the ERR and ERL variables. (For complete explanations of ON ERROR, ERR and ERL, see "Chapter 4. BASIC Commands, Statements, Functions, and Variables.")

This appendix lists all the BASIC error messages with their associated error numbers.

Number	*Message*
1	**NEXT without FOR** The NEXT statement doesn't have a corresponding FOR statement. It may be that a variable in the NEXT statement does not correspond to any previously executed and unmatched FOR statement variable. Fix the program so the NEXT has a matching FOR.
2	**Syntax error** A line contains an incorrect sequence of characters, such as an unmatched parenthesis, a misspelled command or statement, or incorrect punctuation. Or, the data in a DATA statement doesn't match the type (numeric or string) of the variable in a READ statement. When this error occurs, the BASIC program editor automatically displays the line in error. Correct the line or the program.

Number	*Message*
3	**RETURN without GOSUB** A RETURN statement needs a previous unmatched GOSUB statement. Correct the program. You probably need to put a STOP or END statement before the subroutine so the program doesn't "fall" into the subroutine code.
4	**Out of data** A READ statement is trying to read more data than is in the DATA statements. Correct the program so that there are enough constants in the DATA statements for all the READ statements in the program.
5	**Illegal function call** A parameter that is out of range is passed to a system function. The error may also occur as the result of: ● A negative or unreasonably large subscript ● Trying to raise a negative number to a power that is not an integer ● Calling a USR function before defining the starting address with DEF USR ● A negative record number on GET or PUT (file) ● An improper argument to a function or statement ● Trying to list or edit a protected BASIC program ● Trying to delete line numbers which don't exist

Number	*Message*
	Correct the program. Refer to "Chapter 4. Basic Commands, Statements, Functions, and Variables" for information about the particular statement or function.
6	**Overflow** The magnitude of a number is too large to be represented in BASIC's number format. Integer overflow will cause execution to stop. Otherwise, machine infinity with the appropriate sign is supplied as the result and execution continues. To correct integer overflow, you need to use smaller numbers, or change to single- or double-precision variables. **Note:** If a number is too small to be represented in BASIC's number format, we have an *underflow* condition. If this occurs, the result is zero and execution continues without an error.
7	**Out of memory** A program is too large, has too many FOR loops or GOSUBs, too many variables, expressions that are too complicated, or complex painting. You may want to use CLEAR at the beginning of your program to set aside more stack space or memory area.
8	**Undefined line number** A line reference in a statement or command refers to a line which doesn't exist in the program. Check the line numbers in your program, and use the correct line number.

Number	*Message*
9	**Subscript out of range** You used an array element either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts. Check the usage of the array variable. You may have put a subscript on a variable that is not an array, or you may have coded a built-in function incorrectly.
10	**Duplicate Definition** You tried to define the size of the same array twice. This may happen in one of several ways: • The same array is defined in two DIM statements. • The program encounters a DIM statement for an array after the default dimension of 10 is established for that array. • The program sees an OPTION BASE statement after an array has been dimensioned, either by a DIM statement or by default. Move the OPTION BASE statement to make sure it is executed before you use any arrays; or, fix the program so each array is defined only once.
11	**Division by zero** In an expression, you tried to divide by zero, or you tried to raise zero to a negative power. It is not necessary to fix this condition, because the program continues running. Machine infinity with the sign of the

	number being divided is the result of the division; or, positive machine infinity is the result of the exponentiation.
12	**Illegal direct** You tried to enter a statement in direct mode which is invalid in direct mode (such as DEF FN). The statement should be entered as part of a program line.
13	**Type mismatch** You gave a string value where a numeric value was expected, or you had a numeric value in place of a string value. This error may also be caused by trying to SWAP variables of different types, such as single- and double-precision.
14	**Out of string space** BASIC allocates string space dynamically until it runs out of memory. This message means that string variables caused BASIC to exceed the amount of free memory remaining after housecleaning.
15	**String too long** You tried to create a string more than 255 characters long. Try to break the string into smaller strings.
16	**String formula too complex** A string expression is too long or too complex. The expression should be broken into smaller expressions.

Number *Message*

17 **Can't continue**
You tried to use CONT to continue a program that:

- Halted due to an error,
- Was modified during a break in execution, or
- Does not exist

Make sure the program is loaded, and use RUN to run it.

18 **Undefined user function**
You called a function before defining it with the DEF FN statement.

Make sure the program executes the DEF FN statement before you use the function.

19 **No RESUME**
The program branched to an active error trapping routine as a result of an error condition or an ERROR statement. The routine does not have a RESUME statement. (The physical end of the program was encountered in the error trapping routine.)

Be sure to include RESUME in your error trapping routine to continue program execution. You may want to add an ON ERROR GOTO 0 statement to your error trapping routine so BASIC displays the message for any untrapped error.

20 **RESUME without error**
The program has encountered a RESUME statement without having trapped an error. The error trapping routine should only be entered when an error occurs or an ERROR statement is executed.

Number *Message*

You probably need to include a STOP or END statement before the error trapping routine to prevent the program from "falling into" the error trapping code.

22 **Missing operand**
An expression contains an operator, such as * or OR, with no operand following it.

Make sure you include all the required operands in the expression.

23 **Line buffer overflow**
You tried to enter a line that has too many characters.

Separate multiple statements on the line so they are on more than one line. You might also use string variables instead of constants where possible.

24 **Device Timeout**
BASIC did not receive information from an input/output device within a predetermined amount of time. In Cassette BASIC, this only occurs while the program is trying to read from the cassette or write to the printer.

For communications files, this message indicates that one or more of the signals tested with OPEN "COM... was not found in the specified period of time.

Retry the operation.

25 **Device Fault**
A hardware error indication was returned by an interface adapter.

In Cassette BASIC, this only occurs when a fault status is returned from the printer interface adapter.

Number *Message*

25 (cont.) This message may also occur when transmitting data to a communications file. In this case, it indicates that one or more of the signals being tested (specified on the OPEN "COM... statement) was not found in the specified period of time.

26 **FOR without NEXT**
A FOR was encountered without a matching NEXT. That is, a FOR loop was active when the physical end of the program was reached.

Correct the program so it includes a NEXT statement.

27 **Out of Paper**
The printer is out of paper, or the printer is not switched on.

You should insert paper (if necessary), verify that the printer is properly connected, and make sure that the power is on; then, continue the program.

29 **WHILE without WEND**
A WHILE statement does not have a matching WEND. That is, a WHILE was still active when the physical end of the program was reached.

Correct the program so that each WHILE has a corresponding WEND.

30 **WEND without WHILE**
A WEND is encountered before a matching WHILE was executed.

Correct the program so that there is a WHILE for each WEND.

50 **FIELD overflow**
A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file in the OPEN statement. Or, the end of the FIELD buffer is encountered while doing sequential I/O (PRINT #, WRITE #, INPUT #) to a random file.

Check the OPEN statement and the FIELD statement to make sure they correspond. If you are doing sequential I/O to a random file, make sure that the length of the data read or written does not exceed the record length of the random file.

51 **Internal error**
An internal malfunction occurred in BASIC.

Recopy your diskette. Check the hardware and retry the operation. If the error reoccurs, report to your computer dealer the conditions under which the message appeared.

52 **Bad file number**
A statement uses a file number of a file that is not open, or the file number is out of the range of possible file numbers specified at initialization. Or, the device name in the file specification is too long or invalid, or the filename was too long or invalid.

Make sure the file you wanted was opened and that the file number was entered correctly in the statement. Check that you have a valid file specification (refer to "Naming Files" in Chapter 3 for information on file specifications).

Number *Message*

53 **File not found**
A LOAD, KILL, NAME, FILES, or OPEN references a file that does not exist on the diskette in the specified drive.

Verify that the correct diskette is in the drive specified, and that the file specification was entered correctly. Then retry the operation.

54 **Bad file mode**
You tried to use PUT or GET with a sequential file or a closed file; or to execute an OPEN with a file mode other than input, output, append, or random.

Make sure the OPEN statement was entered and executed properly. GET and PUT require a random file.

This error also occurs if you try to merge a file that is not in ASCII format. In this case, make sure you are merging the right file. If necessary, load the program and save it again using the A option.

55 **File already open**
You tried to open a file for sequential output or append, and the file is already opened; or, you tried to use KILL on a file that is open.

Make sure you only execute one OPEN to a file if you are writing to it sequentially. Close a file before you use KILL.

57 **Device I/O Error**
An error occurred on a device I/O operation. DOS cannot recover from the error.

When receiving communications data, this error can occur from overrun, framing, break, or parity errors. When you are receiving data with 7 or less data bits, the eighth bit is turned on in the byte in error.

58 **File already exists**
The filename specified in a NAME statement matches a filename already in use on the diskette.

Retry the NAME command using a different name.

61 **Disk full**
All diskette storage space is in use. Files are closed when this error occurs.

If there are any files on the diskette that you no longer need, erase them; or, use a new diskette. Then retry the operation or rerun the program.

62 **Input past end**
This is an end of file error. An input statement is executed for a null (empty) file, or after all the data in a sequential file was already input.

To avoid this error, use the EOF function to detect the end of file.

This error also occurs if you try to read from a file that was opened for output or append. If you want to read from a sequential output (or append) file, you must close it and open it again for input.

Number	*Message*
63	**Bad record number** In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or equal to zero. Correct the PUT or GET statement to use a valid record number.
64	**Bad file name** An invalid form is used for the filename with BLOAD, BSAVE, KILL, NAME, OPEN, or FILES. Check "Naming Files" in Chapter 3 for information on valid filenames, and correct the filename in error.
66	**Direct statement in file** A direct statement was encountered while loading or chaining to an ASCII format file. The LOAD or CHAIN is terminated. The ASCII file should consist only of statements preceded by line numbers. This error may occur because of a line feed character in the input stream. Refer to "Appendix D. Converting Programs to IBM Personal Computer BASIC."
67	**Too many files** An attempt is made to create a new file (using SAVE or OPEN) when all directory entries on the diskette are full, or when the file specification is invalid. If the file specification is okay, use a new formatted diskette and retry the operation.
68	**Device Unavailable** You tried to open a file to a device which doesn't exist. Either you do not have the hardware to support the device (such as

printer adapters for a second or third printer), or you have disabled the device. (For example, you may have used **/C:0** on the BASIC command to start Disk BASIC. That would disable communications devices.)

Make sure the device is installed correctly. If necessary, enter the command:

```
SYSTEM
```

This returns you to DOS where you can re-enter the BASIC command.

69 **Communication buffer overflow**
A communication input statement was executed, but the input buffer was already full.

You should use an ON ERROR statement to retry the input when this condition occurs. Subsequent inputs attempt to clear this fault unless characters continue to be received faster than the program can process them. If this happens there are several possible solutions:

- Increase the size of the communications buffer using the **/C:** option when you start BASIC.
- Implement a "hand-shaking" protocol with the other computer to tell it to stop sending long enough so you can catch up. (See the example in "Appendix F. Communications.")
- Use a lower baud rate to transmit and receive.

Number	*Message*
70	**Disk Write Protect** You tried to write to a diskette that is write-protected. Make sure you are using the right diskette. If so, remove the write protection, then retry the operation. This error may also occur because of a hardware failure.
71	**Disk not Ready** The diskette drive door is open or a diskette is not in the drive. Place the correct diskette in the drive and continue the program.
72	**Disk Media Error** The controller attachment card detected a hardware or media fault. Usually, this means that the diskette has gone bad. Copy any existing files to a new diskette and re-format the bad diskette. If formatting fails, the diskette should be discarded.
73	**Advanced Feature** Your program used an Advanced BASIC feature while you were using Disk BASIC. Start Advanced BASIC and rerun your program.
—	**Unprintable error** An error message is not available for the error condition which exists. This is usually caused by an ERROR statement with an undefined error code. Check your program to make sure you handle all error codes which you create.

Appendix B. BASIC Diskette Input and Output

This appendix describes procedures and special considerations for using diskette input and output. It contains lists of the commands and statements that are used with diskette files, and explanations of how to use them. Several sample programs are included to help clarify the use of data files on diskette. If you are new to BASIC or if you're getting diskette-related errors, read through these procedures and program examples to make sure you're using all the diskette statements correctly.

You may also want to refer to the *IBM Personal Computer Disk Operating System* manual for other information on handling diskettes and diskette files.

> **Note:** Most of the information in this appendix about program files and sequential files applies to cassette I/O as well. The cassette cannot be opened in random mode, however.

Specifying Filenames

Filenames for diskette files must conform to DOS naming conventions in order for BASIC to be able to read them. Refer to "Naming Files" in Chapter 3 to be sure you are specifying your diskette files correctly.

Commands for Program Files

The commands which you can use with your BASIC program files are listed below, with a quick description. For more detailed information on any of these commands, refer to "Chapter 4. BASIC Commands, Statements, Functions, and Variables."

SAVE **filespec** [,A]
: Writes to diskette the program that is currently residing in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, BASIC uses a compressed binary format.)

LOAD **filespec** [,R]
: Loads the program from diskette into memory. Optional 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.

RUN filespec [,R]

RUN *filespec* loads the program from diskette 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.

MERGE filespec

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

KILL filespec

Deletes the file from the diskette.

NAME filespec AS filename

Changes the name of a diskette file.

Protected Files

If you wish to save a program in an encoded binary format, use the **P** (protect) option with the SAVE command. For example:

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

A program saved this way cannot be listed, saved, or edited. Since you cannot "unprotect" such a program, you may also want to save an unprotected copy of the program for listing and editing purposes.

Diskette Data Files - Sequential and Random I/O

Two types of diskette data files may be created and accessed by a BASIC program: sequential files and random access files.

Sequential Files

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 sequentially, one item after another, in the order that each item is sent. Each item is read back in the same way, from the first item in the file, to the last item.

The statements and functions that are used with sequential files are:

CLOSE	WRITE #
INPUT #	EOF
LINE INPUT #	INPUT$
OPEN	LOC
PRINT #	LOF
PRINT # USING	

Creating and Accessing a Sequential File

To create a sequential file and access the data in the file, include the following steps in your program:

1. Open the file for output or append using the OPEN statement.

2. Write data to the file using the PRINT #, WRITE #, or PRINT # USING statements.

3. To access the data in the file, you must close the file (using CLOSE) and reopen it for input (using OPEN).

4. Use the INPUT# or LINE INPUT# statements to read data from the sequential file into the program.

The following are example program lines that demonstrate these steps.

```
100  OPEN "DATA" FOR OUTPUT AS #1 'step 1
200  WRITE #1,A$,B$,C$             'step 2
300  CLOSE #1                     'step 3
400  OPEN "DATA" FOR INPUT AS #1 'also step 3
500  INPUT #1,X$,Y$,Z$             'step 4
```

The above program could also have been written as follows:

```
100  OPEN "O",#1,"DATA"    'step 1
200  WRITE #1,A$,B$,C$     'step 2
300  CLOSE #1              'step 3
400  OPEN "I",#1,"DATA"    'still step 3
500  INPUT #1,X$,Y$,Z$      'step 4
```

Notice that both ways of writing the OPEN statement yield the same results. Look under "OPEN Statement" in Chapter 4 for details of the syntax of each form of OPEN.

The following program, PROGRAM1, is a short program that creates a sequential file, "DATA", from information you enter at the keyboard.

Program 1

```
1 REM PROGRAM1 - create a sequential file
10 OPEN "DATA" FOR OUTPUT AS #1
20 INPUT "NAME";N$
25 IF N$="DONE" THEN CLOSE: END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 WRITE #1,N$,D$,H$
60 PRINT: GOTO 20
RUN
NAME? MICHELANGELO
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? DONE
Ok
```

Now look at PROGRAM2. It accesses the file "DATA" that was created in PROGRAM1 and displays the name of everyone hired in 1978.

Program 2

```
1 REM PROGRAM2 - accessing a sequential file
10 OPEN "DATA" FOR INPUT AS 1
20 INPUT #1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

PROGRAM2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To avoid getting this error, insert line 15 which uses the EOF function to test for end of file:

```
15 IF EOF(1) THEN CLOSE: END
```

and change line 40 to GOTO 15. The end of file is indicated by a special character in the file. This character has ASCII code 26 (hex 1A). Therefore, you should not put a CHR$(26) in a sequential file.

A program that creates a sequential file can also write formatted data to the diskette with the PRINT # USING statement. For example, the statement:

```
PRINT #1,USING "####.##,";A,B,C,D
```

could be used to write numeric data to diskette without explicit delimiters. The comma at the end of the format string serves to separate the items in the diskette file.

The LOC function, when used with a sequential file, returns the number of records that have been written to or read from the file since it was opened. (A record is a 128-byte block of data.) The LOF function returns the number of bytes allocated to the file. This number is always a multiple of 128 (by rounding upward, if necessary).

Adding Data to a Sequential File

If you have a sequential file residing on diskette and later want to add more data to the end of it, you cannot simply open the file for output and start writing data. As soon as you open a sequential file for output, you destroy its current contents. Instead, you should open the file for APPEND. Refer to "OPEN Statement" in Chapter 4 for details.

Random Files

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. For instance, numbers in random files are usually stored on diskette in binary formats, while numbers in sequential files are stored as ASCII characters. Therefore, in many cases random files require less space on diskette than sequential files.

The biggest advantage to random files is that data can be accessed randomly; that is, anywhere on the diskette. 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.

Records may be any length up to 32767 bytes. The size of a record is not related to the size of a sector on the diskette (512 bytes). BASIC automatically uses all 512 bytes in a sector for information storage. It does this by both blocking records and spanning sector boundaries (that is, part of a record may be at the end of one sector and the other part at the beginning of the next sector).

The statements and functions that are used with random files are:

CLOSE	CVI
FIELD	CVS
GET	LOC
LSET/RSET	LOF
OPEN	MKD$
PUT	MKI$
CVD	MKS$

Creating a Random File

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

1. Open the file for random access. The example which follows to illustrate these steps specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.

3. Use LSET or RSET 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.

4. Write the data from the buffer to the diskette using the PUT statement.

The following lines illustrate these steps:

```
100  OPEN "FILE" AS #1 LEN=32 'step 1
200  FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
                              'step 2
300  LSET N$=X$               'step 3
400  LSET A$=MKS$(AMT)        'still step 3
500  LSET P$=TEL$             'still step 3
600  PUT #1,CODE%             'step 4
```

Note: Do not use a string variable which has been defined in a FIELD statement in an input statement or on the left side of an assignment (LET) statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

Look at PROGRAM3. It takes information that is entered at the keyboard and writes it to a random file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

Program 3

```
1 REM PROGRAM3 - create a random file
10 OPEN "FILE" AS #1 LEN=32
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
35 IF CODE%=99 THEN CLOSE: END
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$: PRINT
70 LSET N$=X$
80 LSET A$=MKS$(AMT)
90 LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

Accessing a Random File

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

1. Open the file for random access.

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file.

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

3. Use the GET statement to move the desired record into the random buffer.

4. 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 values.

The following program lines illustrate these steps:

```
100 OPEN "FILE" AS 1 LEN=32    'step 1
200 FIELD #1 20 AS N$, 4 AS A$, 8 AS P$
                               'step 2
300 GET #1,CODE%               'step 3
400 PRINT N$                   'step 4
500 PRINT CVS(A$)              'still step 4
```

PROGRAM4 accesses the random file "FILE" that was created in PROGRAM3. By entering the two-digit code at the keyboard, the information associated with that code is read from the file and displayed.

Program 4

```
1 REM PROGRAM4 - access a random file
10 OPEN "FILE" AS 1 LEN=32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
35 IF CODE%=99 THEN CLOSE: END
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$: PRINT
80 GOTO 30
```

The LOC function, with random files, returns the "current record number." The current record number is the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1)>50 THEN END
```

ends program execution if the current record number in file #1 is higher than 50.

An Example Program

PROGRAM5 is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part numbers. Lines 690-750 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 180 and line 320) to determine whether an entry already exists for that part number.

Lines 40-120 display the different inventory functions that the program performs. When you type in the desired function number, line 140 branches to the appropriate subroutine.

Program 5

```
10 REM PROGRAM5 - inventory
20 OPEN "inven.dat" AS #1 LEN=39
30 FIELD #1,1 AS F$,30 AS D$,2 AS Q$,2 AS R$,4 AS P$
40 PRINT: PRINT"Options:": PRINT
50 PRINT 1,"Initialize File"
60 PRINT 2,"Create a New Entry"
70 PRINT 3,"Display Inventory for One Part"
80 PRINT 4,"Add to Stock"
90 PRINT 5,"Subtract from Stock"
100 PRINT 6,"List Items Below Reorder Level"
110 PRINT 7,"End Application"
120 PRINT: PRINT: INPUT "Choice";CHOICE
130 IF (CHOICE<1) OR (CHOICE>7) THEN PRINT "Bad Choice Number"
    : GOTO 40
140 ON CHOICE GOSUB 690, 160, 300, 390, 470, 590, 760
150 GOTO 120
160 REM       build new entry
170 GOSUB 670
180 IF ASC(F$)<>255 THEN INPUT "Overwrite";A$:
    IF A$<>"y" AND A$<>"Y" THEN RETURN
190 LSET F$=CHR$(0)
200 INPUT "Description";DESC$
210 LSET D$=DESC$
```

```
220 INPUT "Quantity in stock";Q%
230 LSET Q$=MKI$(Q%)
240 INPUT "Reorder level";R%
250 LSET R$=MKI$(R%)
260 INPUT "Unit price";P
270 LSET P$=MKS$(P)
280 PUT #1,PART%
290 RETURN
300 REM       display entry
310 GOSUB 670
320 IF ASC(F$)=255 THEN PRINT "Null entry": RETURN
330 PRINT USING "Part number ###";PART%
340 PRINT D$
350 PRINT USING "Quantity on hand #####";CVI(Q$)
360 PRINT USING "Reorder level #####";CVI(R$)
370 PRINT USING "Unit price $$##.##";CVS(P$)
380 RETURN
390 REM       add to stock
400 GOSUB 670
410 IF ASC(F$)=255 THEN PRINT"Null entry":RETURN
420 PRINT D$:INPUT "Quantity to add";A%
430 Q%=CVI(Q$)+A%
440 LSET Q$=MKI$(Q%)
450 PUT #1,PART%
460 RETURN
470 REM       remove from stock
480 GOSUB 670
490 IF ASC(F$)=255 THEN PRINT "Null entry": RETURN
500 PRINT D$
510 INPUT "Quantity to subtract";S%
520 Q%=CVI(Q$)
530 IF (Q%-S%)<0 THEN PRINT"Only";Q%;"in stock": GOTO 510
540 Q%=Q%-S%
550 IF Q%=<CVI(R$) THEN PRINT "Quantity now";Q%;
      ", Reorder level";CVI(R$)
560 LSET Q$=MKI$(Q%)
570 PUT #1,PART%
580 RETURN
590 REM       list items below reorder level
600 FOR I=1 TO 100
610 GET #1,I
620 IF ASC(F$)=255 THEN 640
630 IF CVI(Q$)<CVI(R$) THEN PRINT D$;" Quantity";CVI(Q$)
        TAB(50) "Reorder level";CVI(R$)
640 NEXT I
650 RETURN
```

```
660 REM   get part record
670 INPUT "Part number";PART%
680 IF PART%<1 OR PART%>100
      THEN PRINT "Bad part number": GOTO 670
      ELSE GET #1,PART%: RETURN
690 REM initialize file
700 INPUT "Are you sure";B$: IF B$<>"Y" AND B$<>"y"
     THEN RETURN
710 LSET F$=CHR$(255)
720 FOR I=1 TO 100
730 PUT #1,I
740 NEXT I
750 RETURN
760 REM       end application
770 CLOSE: END
```

Performance Hints

- If you do not use random files, specify /S:0 on the BASIC command when you start BASIC. This will save 128 bytes times the number of files specified in the /F: option.
- BASIC sets up three files by default. If you use less than three, set /F:*files* when you start BASIC with the BASIC command. Note that the screen, keyboard, and printer do not count as files unless you explicitly OPEN them.
- Sequential files use a buffer of 128 bytes. Random files also default to a buffer of 128 bytes, but this can be overridden with the /S: option on the BASIC command. There is no advantage to setting /S: to a number greater than the largest record length on any of your random files. However, the combination of a record length of 512 and /S:512 gives improved performance since the diskette sector size is 512 bytes.

If you want to do sequential I/O, but still want improved performance, you can use random files to do "pseudo-sequential" I/O. For example:

```
1 ' example 1A
10 OPEN "ABC" FOR OUTPUT AS #1
20 FOR I=1 TO 3000
30 PRINT #1,"MELH"
40 NEXT
50 CLOSE #1: END
```

This example (1A) uses regular sequential I/O to create a file with 3000 items in it.

```
1 ' example 1B
10 OPEN "ABC" FOR INPUT AS #1
20 OPEN "DEF" FOR OUTPUT AS #2
30 IF EOF(1) THEN CLOSE: END
40 INPUT #1,A$
50 PRINT #2,A$
60 GOTO 30
70 END
```

This example (1B) copies the sequential file "ABC", which we just created, to a file named "DEF".

For the next examples we will perform the same task using "pseudo-sequential" I/O.

```
1 ' example 2A
10 OPEN "PQR" AS #1 LEN=512
15 ON ERROR GOTO 90
20 FOR I=1 TO 3000
30 PRINT #1,"MELH"
40 NEXT
45 PRINT #1,"/eof"
50 ON ERROR GOTO 0: PUT #1: CLOSE
60 END
90 PUT #1: RESUME
```

This example (2A) creates a file with 3000 items using random I/O. This is a "pseudo-sequential" file.

```
1 ' example 2B
10 OPEN "PQR" AS #1 LEN=512
20 OPEN "XYZ" AS #2 LEN=512
30 ON ERROR GOTO 80
40 GET #1
50 INPUT #1,A$
60 PRINT #2,A$
70 IF A$<>"/eof" THEN 50 ELSE
       ON ERROR GOTO 0: PUT #2: CLOSE: END
80 IF ERL=50 THEN GET #1: RESUME
       ELSE PUT #2: RESUME
```

This final example copies the "pseudo-sequential" file created in the previous example to a new "pseudo-sequential" file named "XYZ". It takes about half as long to run as the example using sequential I/O.

The technique used in these examples involves detection of the "FIELD overflow" error (error 50) and doing the appropriate GET or PUT to purge the buffer (line 90 in example 2A and line 80 in example 2B). Note also that a dummy end of file must be written ("/eof" in the example) and checked for during input. Also, the INPUT and PRINT statements use only single variables, rather than a list of variables.

This technique is useful only when you have more than one file open at a time.

NOTES

Appendix C. Machine Language Subroutines

This appendix describes how BASIC interfaces with machine language subroutines. In particular, it describes:

- How to allocate memory for the subroutines
- How to get the machine language subroutine into memory
- How to call the subroutine from BASIC and pass parameters to it

This appendix is intended to be used by an experienced machine language programmer.

Reference Material

Rector, Russell and Alexy, George. *The 8086 Book*. Osborne/McGraw-Hill, Berkeley, California, 1980. (includes the 8088)

Intel Corporation Literature Department. *The 8086 Family User's Manual*, 9800722. 3065 Bowers Avenue, Santa Clara, CA 95051.

IBM Corporation Personal Computer library. *Macro-Assembler*. Boca Raton, FL 33432.

IBM Corporation Personal Computer library. *Technical Reference*. Boca Raton, FL 33432.

Setting Memory Aside for Your Subroutines

BASIC normally uses all memory available from its starting location up to a maximum of 64K-bytes. This BASIC workarea contains your BASIC program and data, along with the interpreter workarea and BASIC's stack. You may allocate memory space for machine language subroutines either inside or outside of this BASIC 64K workarea. Where you decide to put the routines depends on the total amount of available memory and the size of the applications to be loaded.

Your system needs more than 64K-bytes of memory if you want to put your machine language subroutines outside BASIC's 64K workarea. If you are using Disk or Advanced BASIC, DOS takes up approximately 12K-bytes, and BASIC takes up another 10K-bytes, so you need at least a 96K-byte system in order for there to be room outside the BASIC workarea for the machine language subroutines.

Outside the BASIC Workarea: If your system has enough memory that you can put your subroutines outside the BASIC 64K-byte workarea, you don't have to do anything to reserve that area. You use the DEF SEG statement to address the external subroutine area outside the BASIC workarea.

For example, in a 96K-byte system, to specify an address in the upper 4K-bytes of memory, you could use:

```
110 DEF SEG=&H1700
```

This statement specifies a segment starting at hexadecimal location 17000 (92K).

Inside the BASIC Workarea: In order to keep BASIC from writing over your subroutines in memory, use either:

- The CLEAR statement, which is available in all versions of BASIC
- The /M: option on the BASIC command to start Disk and Advanced BASIC from DOS

Only the highest memory locations can be set aside for subroutines. For example, to reserve the highest 4K-byte area of BASIC's 64K-byte workarea for your machine language subroutines, you could use:

```
10 CLEAR ,&HF000
```

or start BASIC with the DOS command:

```
BASIC /M:&HF000
```

Either of these statements restricts the size of the BASIC workarea to hex F000 (60K) bytes, so you can use the uppermost 4K-bytes for machine language subroutines.

Getting the Subroutine Code into Memory

The following are offered as suggestions as to how machine language subroutines can be loaded. We don't describe all possible situations.

Two common ways to get a machine language program into memory are:

- Poking it into memory from your BASIC program
- Loading it from a file on diskette or cassette

Poking a Subroutine into Memory

You can code relatively short subroutines in machine language and use the POKE statement to put the code into memory. In this way, the subroutine actually becomes a part of your BASIC program. One way to do this is:

1. Determine the machine code for your subroutine.

2. Put the hex value (&Hxx format) of each byte of the code into DATA statements.

3. Execute a loop which reads each data byte, and then pokes it into the area you've selected for the subroutine (see the preceding discussion).

4. After the loop is complete, the subroutine is loaded. If you are going to call the subroutine using the USR function, then you must execute a DEF USR statement to define the entry address of the subroutine; if you are going to call the subroutine using the CALL statement, you must set the value of the subroutine variable to the subroutine's entry address.

For example:

```
Ok
1Ø DEFINT A-Z
2Ø DEF SEG=&H17ØØ
3Ø FOR I=Ø TO 21
4Ø READ J
5Ø POKE I,J
6Ø NEXT
7Ø SUBRT=Ø
8Ø A=2:B=3:C=Ø
9Ø CALL SUBRT(A,B,C)
1ØØ PRINT C
11Ø END
12Ø DATA &H55,&H8B,&HEC,&H8B,&H76,&HØA
13Ø DATA &H8B,&HØ4,&H8B,&H76,&HØ8
14Ø DATA &HØ3,&HØ4,&H8B,&H7E,&HØ6
15Ø DATA &H89,&HØ5,&H5D,&HCA,&HØ6,&HØØ
RUN
 5
Ok
```

Loading the Subroutine from a File

You use the BASIC BLOAD command to load a memory image file directly into memory. The memory image can be a machine language subroutine which was saved using the BSAVE command. Of course, that leads to the question of how the subroutine got there in the first place. The machine language subroutine may be an executable file which was created by the linker from DOS, and which was placed into memory using DEBUG. DEBUG and the linker are explained in the *IBM Personal Computer Disk Operating System* manual.

The following is a suggested way to use BLOAD to get such a machine language subroutine into memory:

1. Use the linker to produce an .EXE file of your routine (let's call it ASMROUT.EXE) so it will load at the HIGH end of memory.

2. Load BASIC under DEBUG by entering:

   ```
   DEBUG BASIC.COM
   ```

3. Display the registers (use the R command) to find out where BASIC was put in memory. Record the values contained in the registers (CS, IP, SS, SP, DS, ES) for later reference.

4. Use DEBUG to load the .EXE file (your subroutine) into HIGH memory, where it will overlay the transient portion of COMMAND.COM.

   ```
   N ASMROUT.EXE
   L
   ```

5. Display the registers (use the R command) to find out where the subroutine was placed in memory. Record the values contained in the CS and IP registers for later use.

6. Reset the registers (use the R command) back to the values they contained when BASIC.COM was originally loaded, using the values noted in step 3.

7. Use the G command to branch to the BASIC entry point and to set breakpoints (if desired) in the machine language subroutine.

8. When BASIC prompts, load your BASIC application program and edit the DEF SEG and either the DEF USR statement or the value of the CALL variable to correspond with the location of the subroutine as determined when you loaded the subroutine in step 5.

   - Use the previously recorded value in the CS register for DEF SEG
   - Use the previously recorded value in the IP register for the DEF USR or the variable value of the CALL

9. In direct mode in BASIC, enter a BSAVE command to save the subroutine area. Use the starting location defined by the CS and IP registers when the subroutine was loaded in step 5, and the code length printed on the assembler listing or LINK map. (Refer to "BSAVE Command" in Chapter 4.)

10. Edit your BASIC application program so it contains a BLOAD statement after the DEF SEG that sets the proper value of CS for the subroutine.

Note: If the machine language routine is self-relocatable, BLOAD can be used to put the subroutine some place other than where the linker originally placed it. If you make such a change, be sure to make a corresponding change to the DEF SEG statement associated with the call so that BASIC can find the subroutine at execution time.

Some suggestions for alternate locations for the subroutine are:

- An unused screen buffer
- An unused file buffer (located with VARPTR(#*f*))
- A string variable area located with VARPTR(*stringvar*)

(See 'BLOAD Command" and "VARPTR Function" in Chapter 4.)

11. Save the resulting modified BASIC application.

Some Notes on Using DEBUG with BASIC: When you run BASIC under DEBUG, BASIC is loaded after DEBUG in memory, so DEBUG is not written over if you load a large BASIC program. If you set breakpoints in your machine language subroutine, they return you to DEBUG. The SYSTEM command also returns you from BASIC to DEBUG.

Calling the Subroutine from Your BASIC Program

All versions of BASIC have two ways to call machine language subroutines: the USR function, and the CALL statement. This section describes the use of both USR and CALL.

Common Features of CALL and USR

Whether you call your machine language subroutines with CALL or with the USR function, you must keep the following things in mind:

Entering the Subroutine

- At entry, the segment registers DS, ES, and SS are all set to the same value, the address of BASIC's data space (the default for DEF SEG).

- At entry, the code segment register, CS, contains the current value specified in the latest DEF SEG. If DEF SEG has not been specified, or if the latest DEF SEG did not specify an override value, the value in CS is the same as in the other three segment registers.

String Arguments

- If an input argument is a string, the value received in the argument is the address of a three-byte area called the *string descriptor*:

  1. Byte 0 of the string descriptor contains the length of the string (0 to 255).

  2. Byte 1 of the string descriptor contains the lower 8 bits of the offset of the string in BASIC's data space.

3. Byte 2 of the string descriptor contains the higher 8 bits of the offset of the string in BASIC's data space.

The string itself is pointed to by the last two bytes of the string descriptor.

Warning:
The subroutine must not change the contents of any of the three bytes of the *string descriptor.*

The subroutine may change the *content* of the string itself, but not its *length*.

If the subroutine changes a string, be aware that this may *modify your program.* The following example may change the string "TEXT"in the BASIC program.

```
A$ = "TEXT"
CALL SUBRT(A$)
```

However, the next example does not modify the program, because the string concatenation causes BASIC to copy the string into the string space where it may be safely changed without affecting the original text.

```
A$ = "BASIC" +""
CALL SUBRT(A$)
```

Returning from the Subroutine

- The return to BASIC must be by an inter-segment RET instruction. (The subroutine is a FAR procedure.)

- At exit, all segment registers and the stack pointer, SP, must be restored. All other registers (and flags) may be altered.

- The stack pointer, at entry, indicates a stack that has only 16 bytes (eight words) available for use by the subroutine. If more stack space is needed, the subroutine must set up its own stack segment and stack pointer. You should make sure that the location of the current stack is recorded so its pointer can be restored just prior to return.
- If interrupts were disabled by the subroutine, they should be enabled prior to return.

CALL Statement

Machine language subroutines may be called using the BASIC CALL statement. The format of the CALL statement is:

CALL *numvar* [(*variable list*)]

numvar	is the name of a numeric variable. Its value is the offset, from the segment set by DEF SEG, that is the starting point in memory of the subroutine being called.
variable list	contains the variables, separated by commas, that are to be passed as arguments to the routine. (The arguments cannot be constants.)

Execution of a CALL statement causes the following:

1. For each variable in the variable list, the variable's location is pushed onto the stack. The location is specified as a two-byte offset into BASIC's data segment (the default DEF SEG).

2. The return address specified in the CS register and the offset are pushed onto the stack.

3. Control is transferred to the machine language routine using the segment address specified in the last DEF SEG statement and the offset specified by the value of *numvar*.

Notes for the CALL Statement

- You can return values to BASIC through the arguments by changing the values of the variables in the argument list.

- If the argument is a string, the offset for the argument points to the three-byte *string descriptor* as explained previously.

- The called routine must know how many arguments were passed. Parameters are referenced by adding a positive offset to BP after the called routine moves the current stack pointer into BP. The first instructions in the subroutine should be:

```
PUSH BP      ;SAVE BP
MOV BP,SP    ;MOVE SP TO BP
```

The offset into the stack of any one particular argument is calculated as follows:

offset from BP = 2*(n-m)+6

where:

n is the total number of arguments passed.

m is the position of the specific argument in the argument list of the BASIC CALL statement (m may range from 1 to n).

Example: The following example adds the values in A% and B% and stores the result in C%:

The following statements are in BASIC:

```
100 A%=2: B%=3
200  DEF SEG=&H27E0
250  BLOAD "SUBRT.EXE",0
300  SUBRT=0
400  CALL SUBRT (A%,B%,C%)
500 PRINT C%
```

> **Note:** Line 200 sets the segment to location hex 27E00. SUBRT is set to 0 so that the call to SUBRT executes the subroutine at location &H27E00.

The following statements are in IBM Personal Computer Macro-Assembler source code:

```
CSEG    SEGMENT
        ASSUME   CS:CSEG
SUBRT   PROC     FAR
        PUSH BP          ;SAVE BP
        MOV BP,SP        ;SET BASE PARM LIST
        MOV SI,[BP]+10   ;GET ADDR PARM A
        MOV AX,[SI]      ;GET VALUE OF A
        MOV SI,[BP]+8    ;GET ADDR PARM B
        ADD AX,[SI]      ;ADD VALUE B TO REG
        MOV DI,[BP]+6    ;GET ADDR PARM C
        MOV [DI],AX      ;PASS BACK SUM
        POP BP           ;RESTORE BP
        RET 6            ;FAR RETURN TO BASIC
SUBRT   ENDP
CSEG    ENDS
        END
```

> **Note:** When you call a routine using the CALL statement, the routine must return with a RET *n*, where *n* is 2 times the number of arguments in the variable list. This is necessary to adjust the stack to the point at the start of the calling sequence.

As another example:

```
10 DEFINT A-Z
100 DEF SEG=&H1800
110 BLOAD "SUBRT.EXE",0
120 SUBRT=0
130 CALL SUBRT (A,B$,C)
```

The following sequence of Macro-Assembler code shows how the arguments (including the address of a string descriptor) are passed and accessed, and how the result is stored in variable C:

```
PUSH   BP          ;SAVE BP
MOV    BP,SP       ;GET CURRENT STK POSITION INTO BP
MOV    BX,[BP]+8   ;GET ADDR OF B$ STRING DESCRIPTOR
MOV    CL,[BX]     ;GET LENGTH OF B$ INTO CL
MOV    DX,1[BX]    ;GET ADDR OF B$ TEXT INTO DX
  .
  .
  .
MOV    SI,[BP]+10  ;GET ADDR OF A INTO SI
MOV    DI,[BP]+6   ;GET ADDR OF C INTO DI
MOVS   WORD        ;STORE VARIABLE A INTO C
POP    BP          ;RESTORE BP
RET    6           ;RESTORE STACK, RETURN
END
```

Warning:
It is entirely up to you to make sure that the arguments in the CALL statement match in number, type, and length with the arguments expected by the subroutine.

In the preceding example, the instruction **MOVS WORD** copies only two bytes because variables A and C are integers. However, if A and C are single-precision, four bytes must be copied; if A and C are double-precision, eight bytes must be copied.

USR Function Calls

The other way to call machine language subroutines from BASIC is with the USR function. The format of the USR function is:

USR[*n*](*arg*)

n	must be a single digit in the range 0 through 9.
arg	is any numeric expression or a string variable name.

n specifies which USR routine is being called, and corresponds with the digit supplied in the DEF USR statement for that routine. If *n* is omitted, USR0 is assumed. The address specified in the DEF USR statement determines the starting address of the subroutine. Even if the subroutine does not require an argument, a dummy argument must still be supplied.

When the USR function is called, register AL contains a value that specifies the type of argument that was supplied. The value in AL will be one of the following:

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

If the argument is a string, the DX register points to the three-byte string descriptor. (See "Common Features of CALL and USR," described previously.)

If the argument is a number and not a string, the value of the argument is placed in the Floating Point Accumulator (FAC), which is an eight-byte area in BASIC's data space. In this case, the BX register contains the offset within the BASIC data space to the fifth byte of the eight-byte FAC. For the following examples, assume that the FAC is in bytes hex 49F through hex 4A6; that is, BX contains hex 4A3:

If the argument is an integer:

- Hex 4A4 contains the upper 8 bits of the argument.

- Hex 4A3 contains the lower 8 bits of the argument.

If the argument is a single-precision number:

- Hex 4A6 contains the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa. Hex 4A5 contains the highest 7 bits of the mantissa with the leading 1 suppressed (implied). Bit 7 is the sign of the number (0 = positive; 1 = negative).

- Hex 4A4 contains the middle 8 bits of the mantissa.

- Hex 4A3 contains the lowest 8 bits of the mantissa.

If the argument is a double-precision number:

- Hex 4A3 through hex 4A6 are the same as described under single-precision floating-point number in the preceding paragraph.

- Hex 49F through Hex 4A2 contain four more bytes of the mantissa (hex 49F contains the lowest 8 bits).

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, a numerical argument of the function, regardless of its type, may be forced to an integer value by calling the FRCINT routine to get the integer equivalent of the argument placed into register BX.

If the value being returned by the function is to be an integer, place the resulting value into the BX register. Then make a call to MAKINT just prior to the inter-segment return. This passes the value back to BASIC by placing it into the FAC.

The methods for accessing FRCINT and MAKINT are shown in the following example:

```
100 DEF SEG=&H1800
120 BLOAD "SUBRT.EXE",0
130 DEF USR0=0
140 X = 5 'Note that X is single-precision
150 Y = USR0(X)
160 PRINT Y
```

At location 1800:0 (segment:offset), the following Macro-Assembler language routine has been loaded. The routine doubles the argument passed and returns an integer result:

```
RSEG      SEGMENT AT ØF6ØØH ;BASE OF BASIC ROM
          ORG   3       ;OFFSET TO FORCE INTEGER
  FRCINT  LABEL FAR
          ORG   7       ;OFFSET TO MAKE INTEGER
  MAKINT  LABEL FAR
RSEG     ENDS
   .
   .
   .
CSEG      SEGMENT
USRPRG    PROC  FAR     ;ENTRY POINT
          CALL  FRCINT  ;FORCE ARG IN FAC INTO [BX]
          ADD   BX,BX   ;[BX] = [BX] * 2
          CALL  MAKINT  ;PUT INT RSLT IN BX INTO FAC
          RET           ;INTER-SEGMENT RETURN TO BASIC
USRPRG    ENDP
CSEG      ENDS
```

Note: FRCINT and MAKINT perform inter-segment returns. You should make sure that the calls to FRCINT and MAKINT are defined by a FAR procedure.

NOTES

Appendix D. Converting Programs to IBM Personal Computer BASIC

Since IBM Personal Computer BASIC is very similar to many microcomputer BASICs, the IBM Personal Computer will support programs written for a wide variety of microcomputers. If you have programs written in a BASIC other than IBM Personal Computer BASIC, some minor adjustments may be necessary before running them with IBM Personal Computer BASIC. Here are some specific things to look for when converting BASIC programs.

File I/O

In IBM Personal Computer BASIC, you read and write information to a file on diskette or cassette by opening the file to associate it with a particular file number; then using particular I/O statements which specify that file number. I/O to diskette and cassette files is implemented differently in some other BASICs. Refer to the section in Chapter 3 called "Files," and to "OPEN Statement" in Chapter 4 for more specific information.

Also, in IBM Personal Computer BASIC, random file records are automatically blocked as appropriate to fit as many records as possible in each sector.

Graphics

How you draw on the screen varies greatly between different BASICs. Refer to the discussion of graphics in Chapter 3 for specific information about IBM Personal Computer graphics.

IF...THEN

The IF statement in IBM Personal Computer BASIC contains an optional ELSE clause, which is performed when the expression being tested is false. Some other BASICs do not have this capability. For example, in another BASIC you may have:

```
10 IF A=B THEN 30
20 PRINT "NOT EQUAL" : GOTO 40
30 PRINT "EQUAL"
40 REM CONTINUE
```

This sequence of code will still function correctly in IBM Personal Computer BASIC, but it may also be conveniently recoded as:

```
10 IF A=B THEN PRINT "EQUAL" ELSE PRINT "NOT EQUAL"
20 REM CONTINUE
```

IBM Personal Computer BASIC also allows multiple statements in both the THEN and ELSE clauses. This may cause a program written in another BASIC to perform differently. For example:

```
10 IF A=B THEN GOTO 100 : PRINT "NOT EQUAL"
20 REM CONTINUE
```

In some other BASICs, if the test A=B is false, control branches to the next *statement*; that is, if A is not equal to B, "NOT EQUAL" is printed. In IBM Personal Computer BASIC, both GOTO 100 and PRINT "NOT EQUAL" are considered to be part of the THEN clause of the IF statement. If the test is false, control continues with the next program *line*; that is, to line 20 in this example. PRINT "NOT EQUAL" can never be executed.

This example can be recoded in IBM Personal Computer BASIC as:

```
10 IF A=B THEN 100 ELSE PRINT "NOT EQUAL"
20 REM CONTINUE
```

Line Feeds

In other BASICs, when you enter a line feed, a line feed character is actually inserted into the text. On the IBM Personal Computer, entering a line feed will pad the rest of the display line with spaces — it does not insert the line feed character. If you try to load a program with line feed characters in it, you will get a "Direct statement in file" error.

Logical Operations

In IBM Personal Computer BASIC, logical operations (NOT, AND, OR, XOR, IMP, and EQV) are performed bit-by-bit on integer operands to produce an integer result. In some other BASICs, the operands are considered to be simple "true" (non-zero) or "false" (zero) values, and the result of the operation is either true or false. As an example of this difference, consider this small program:

```
10 A=9: B=2
20 IF A AND B THEN PRINT "BOTH A AND B ARE TRUE"
```

This example in another BASIC will perform as follows: A is non-zero, so it is true; B is also non-zero, so it is also true; because both A and B are true, A AND B is true, so the program prints **BOTH A AND B ARE TRUE.**

However, IBM Personal Computer BASIC calculates it differently: A is 1001 in binary form, and B is 0010 in binary form, so A AND B (calculated bit-by-bit) is 0000, or zero; zero indicates false, so the message is *not* printed, and the program continues with the next line.

This can affect not only tests made in IF statements, but calculations as well. To get similar results, recode logical expressions like the following:

```
10 A=9: B=2
20 IF (A<>0) AND (B<>0)
      THEN PRINT "BOTH A AND B ARE TRUE"
```

MAT Functions

Programs using the MAT functions available in some BASICs must be rewritten using FOR...NEXT loops to execute properly.

Multiple Assignments

Some BASICs allow statements of the form:

```
10 LET B=C=0
```

to set B and C equal to zero. IBM Personal Computer BASIC would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0. Instead, convert this statement to two assignment statements:

```
10 C=0:B=0
```

Multiple Statements

Some BASICs use a backslash (\) to separate multiple statements on a line. With IBM Personal Computer BASIC, be sure all statements on a line are separated by a colon (:).

PEEKs and POKEs

Many PEEKs and POKEs are dependent on the particular computer you are using. You should examine the *purpose* of the PEEKs and POKEs in a program in another BASIC, and translate the statement so it performs the same function on the IBM Personal Computer.

Relational Expressions

In IBM Personal Computer BASIC, the value returned by a relational expression, such as A>B, is either -1, indicating the relation is true, of 0, indicating the relation is false. Some other BASICs return a positive 1 to indicate true. If you use the value of a relational expression in an arithmetic calculation, the results are likely to be different from what you want.

Remarks

Some BASICs allow you to add remarks to the end of a line using the exclamation point (!). Be sure to change this to a single quote (') when converting to IBM Personal Computer BASIC.

Rounding of Numbers

IBM Personal Computer BASIC rounds single- or double-precision numbers when it requires an integer value. Many other BASICs truncate instead. This can change the way your program runs, because it affects not only assignment statements (for example, I%=2.5 results in I% equal to 3), but also affects function and statement evaluations (for example, TAB(4.5) goes to the fifth position, A(1.5) is the same as A(2), and X=11.5 MOD 4 will result in a value of 0 for X). Note in particular that rounding may cause IBM Personal Computer BASIC to select a different element from an array than another BASIC — possibly one that is out of range!

Sounding the Bell

Some BASICs require PRINT CHR$(7) to send an ASCII bell character. In IBM Personal Computer BASIC, you may replace this statement with BEEP, although it is not required.

String Handling

String Length: Since strings in IBM Personal Computer BASIC are all variable length, you should delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the IBM Personal Computer BASIC statement DIM A$(J).

Concatenation: Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for IBM Personal Computer BASIC string concatenation.

Substrings: In IBM Personal Computer BASIC, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from positin I to position J, must be changed as follows:

Other BASIC	IBM Personal Computer BASIC
`X$=A$(I)`	`X$=MID$(A$,I,1)`
`X$=A$(I,J)`	`X$=MID$(A$,I,J-I+1)`

If the substring reference is on the left side of an assignment and X$ is used to replace characters in A$, convert as follows:

Other BASIC	IBM Personal Computer BASIC
`A$(I)=X$`	`MID$(A$,I,1)=X$`
`A$(I,J)=X$`	`MID$(A$,I,J-I+1)=X$`

Use of Blanks

Some BASICs allow statements with no separation of keywords:

```
20FORI=1TOX
```

With IBM Personal Computer BASIC be sure all keywords are separated by a space:

```
20 FOR I=1 TO X
```

Other

The BASIC language on another computer may be different from the IBM Personal Computer BASIC in other ways than those listed here. You should become familiar with IBM Personal Computer BASIC as much as possible in order to be able to appropriately convert any function you may require.

NOTES

Appendix E. Mathematical Functions

Functions that are not intrinsic to IBM Personal Computer BASIC may be calculated as follows.

Function	Equivalent
Logarithm to base B	LOGB(X) = LOG(X)/LOG(B)
Secant	SEC(X) = 1/COS(X)
Cosecant	CSC(X) = 1/SIN(X)
Cotangent	COT(X) = 1/TAN(X)
Inverse sine	ARCSIN(X) = ATN(X/SQR(1-X*X))
Inverse cosine	ARCCOS(X) = 1.570796 -ATN(X/SQR(1-X*X))
Inverse secant	ARCSEC(X) = ATN(SQR(X*X-1)) +(X<0)*3.141593
Inverse cosecant	ARCCSC(X) = ATN(1/SQR(X*X-1)) +(X<0)*3.141593
Inverse cotangent	ARCCOT(X) = 1.57096-ATN(X)
Hyperbolic sine	SINH(X) = (EXP(X)-EXP(-X))/2
Hyperbolic cosine	COSH(X) = (EXP(X)+EXP(-X))/2
Hyperbolic tangent	TANH(X) = (EXP(X)-EXP(-X)) /(EXP(X)+EXP(-X))
Hyperbolic secant	SECH(X) = 2/(EXP(X)+EXP(-X))
Hyperbolic cosecant	CSCH(X) = 2/(EXP(X)-EXP(-X))
Hyperbolic cotangent	COTH(X) = (EXP(X)+EXP(-X)) /(EXP(X)-EXP(-X))
Inverse hyperbolic sine	ARCSINH(X) = LOG(X+SQR(X*X+1))
Inverse hyperbolic cosine	ARCCOSH(X) = LOG(X+SQR(X*X-1))
Inverse hyperbolic tangent	ARCTANH(X) = LOG((1+X)/(1-X))/2
Inverse hyperbolic secant	ARCSECH(X) = LOG((1+SQR(1-X*X))/X)
Inverse hyperbolic cosecant	ARCCSCH(X) = LOG((1+SGN(X) *SQR(1+X*X))/X)
Inverse hyperbolic cotangent	ARCCOTH(X) = LOG((X+1)/(X-1))/2

If you use these functions, a good way to code them would be with the DEF FN statement. For example, instead of coding the formula for inverse hyperbolic sine each time you need it, you could code:

```
DEF FNARCSINH(X) = LOG(X+SQR(X*X+1))
```

in one place, then refer to it as

```
FNARCSINH(Y)
```

each time you need it.

Appendix F. Communications

This appendix describes the BASIC statements required to support RS232 asynchronous communication with other computers and peripherals.

Opening a Communications File

OPEN "COM... allocates a buffer for I/O in the same fashion as OPEN for diskette files. Refer to "OPEN "COM... Statement" in Chapter 4.

Communication I/O

Since each communications adapter is opened as a file, all input/output statements that are valid for diskette files are valid for communications.

Communications sequential input statements are the same as those for diskette files. They are:

```
INPUT #
LINE INPUT #
INPUT$
```

Communications sequential output statements are the same as those for diskette files, and are:

```
PRINT #
PRINT # USING
WRITE #
```

Refer to the INPUT and PRINT sections for details of coding syntax and usage.

GET and PUT for Communications Files

GET and PUT are only slightly different for communications files than for diskette files. They are used for fixed length I/O from or to the communications file. In place of specifying the record number to be read or written, you specify the number of bytes to be transferred into or out of the file buffer. This number cannot exceed the value set by the LEN option on the OPEN "COM... statement. Refer to the GET and PUT sections in Chapter 4.

I/O Functions

The most difficult aspect of asynchronous communication is being able to process characters as fast as they are received. At rates of 1200 bps or higher, it may be necessary to suspend character transmission from the other computer long enough to "catch up." This can be done by sending XOFF (CHR$(19)) to the other computer and XON (CHR$(17)) when ready to resume. XOFF tells the other computer to stop sending, and XON tells it it can start sending again.

> **Note:** This is a commonly used convention, but it is not universal. It depends on the protocol implemented between you and the other computer or peripheral.

Disk and Advanced BASIC provide three functions which help in determining when an "overrun" condition may occur. These are:

LOC(f)	Returns the number of characters in the input buffer waiting to be read. If the number is greater than 255, LOC returns 255.
LOF(f)	Returns the amount of free space in the input buffer. This is the same as *n*-LOC(f), where *n* is the size of the communictions buffer as set by the /C: option on the BASIC command. The default for *n* is 256.

EOF(f) Returns true (-1) if the input buffer is empty; false (0) if there are any characters waiting to be read.

Note: A "Communication buffer overflow" can occur if a read is attempted after the input buffer is full (that is, when LOF(f) returns 0).

INPUT$ Function

The INPUT$ function is preferred over the INPUT # and LINE INPUT # statements when reading communications files, since all ASCII characters may be significant in communications. INPUT # is least desirable because input stops when a comma (,) or carriage return is seen. LINE INPUT # stops when a carriage return is seen.

INPUT$ allows all characters read to be assigned to a string. INPUT$(*n,f*) will return *n* characters from the #*f* file. The following statements are efficient for reading a communications file:

```
110 WHILE NOT EOF(1)
120 A$=INPUT$(LOC(1),#1)
  .
  .
  .
(process data returned in A$)
  .
  .
  .
190 WEND
```

These statements return the characters in the buffer into A$ and process them, as long as there are characters in the input buffer. If there are more than 255 characters in the buffer, only 255 will be returned at a time to prevent "String overflow." Further, if this is the case, EOF(1) is false and input continues until the input buffer is empty. Simple, concise, and fast.

In order to process characters quickly, you should avoid, if possible, examining every character as you receive it. If you are looking for special characters (such as control characters), you can use the INSTR function to find them in the input string.

An Example Program

The following program enables the IBM Personal Computer to be used as a conventional "dumb" terminal in a full duplex mode. This program assumes a 300 bps line and an input buffer of 256 bytes (the /C: option was not used on the BASIC command).

```
10 REM   dumb terminal example
20 'set screen to black and white text mode
30 '     and set width to 40
40 SCREEN 0,0: WIDTH 40
50 'turn off soft key display; clear screen;
60 '    make sure all files are closed
70 KEY OFF: CLS: CLOSE
80 'define all numeric variables as integer
90 DEFINT A-Z
100 'define true and false
110 FALSE=0: TRUE= NOT FALSE
120 'define the XON, XOFF characters
130 XOFF$=CHR$(19): XON$=CHR$(17)
140 'open communications to file number 1,
150 ' 300 bps, EVEN parity, 7 data bits
160 OPEN "COM1:300,E,7" AS #1
170 'use screen as a file, just for fun
180 OPEN "SCRN:" FOR OUTPUT AS 2
190 'turn cursor on
200 LOCATE ,,1
400 PAUSE=FALSE: ON ERROR GOTO 9000
490 '
```

```
500 'send keyboard input to com line
510 B$=INKEY$: IF B$<>"" THEN PRINT #1,B$;
520 'if no chars in com buffer, check key in
530 IF EOF(1) THEN 510
540 'if buffer more than 1/2 full, then
550 '   set PAUSE flag to say input suspended,
560 '   send XOFF to host to stop transmission
570 IF LOC(1)>128 THEN PAUSE=TRUE: PRINT #1,XOFF$;
580 'read contents of com buffer
590 A$=INPUT$(LOC(1),#1)
600 'get rid of linefeeds to avoid double spaces
610 '   when input displayed on screen
620 LFP=0
630 LFP=INSTR(LFP+1,A$,CHR$(10)) 'look for LF
640 IF LFP>0 THEN MID$(A$,LFP,1)=" ": GOTO 630
650 'display com input, and check for more
660 PRINT #2,A$;: IF LOC(1)>0 THEN 570
670 'if transmission suspended by XOFF,
680 '   resume by sending XON
690 IF PAUSE THEN PAUSE=FALSE: PRINT #1,XON$;
700 'check for keyboard input again
710 GOTO 510
8999 'if error, display error number and retry
9000 PRINT "ERROR NO.";ERR: RESUME
```

Notes on the Program

- "Asynchronous" communication implies character I/O as opposed to line or block I/O. Therefore, all PRINTs (either to communications file or to screen) are terminated with a semicolon (;). This stops the carriage return normally issued at the end of the list of values to be printed.

- Line 90, where all numeric variables are defined as integer, is coded because any program looking for speed optimization should use integer counters in loops where possible.

- Note in line 510 that INKEY$ will return a null string if no character is pending.

Operation of Control Signals

This section contains more detailed technical information that you may need to know to communicate with another computer or peripheral from BASIC.

The output from the Asynchronous Communications Adapter conforms to the EIA RS232-C standard for interface between Data Terminal Equipment (DTE) and Data Communications Equipment (DCE). This standard defines a number of control signals that are transmitted or received by your IBM Personal Computer to control the interchange of data with another computer or peripheral. These signals are DC voltages that are either ON (greater than +3 volts) or OFF (less than -3 volts). See the *IBM Personal Computer Technical Reference* manual for details.

Control of Output Signals with OPEN

When you start BASIC on your IBM Personal Computer, the RTS (Request To Send) and DTR (Data Terminal Ready) lines are held OFF. When an OPEN "COM... statement is performed, both of these lines are normally turned ON. However, you can specify the **RS** option on the OPEN "COM... statement to suppress the RTS signal. The lines stay ON until the communications file is closed (by CLOSE, END, NEW, RESET, SYSTEM, or RUN without the R option). Even if the OPEN "COM... statement fails with an error (as described below), the DTR line (and RTS line, if applicable) is turned ON and stays ON. This allows you to retry the OPEN without having to execute a CLOSE.

Use of Input Control Signals

Normally, if either the CTS (Clear To Send) or DSR (Data Set Ready) lines are OFF, then an OPEN "COM... statement will not execute. After one second, BASIC will return with a "Device Timeout" error (error code 24). The Carrier Detect (sometimes called Receive Line Signal Detect) can be either ON or OFF; it has no effect on the operation of the program.

However, you can specify how you want these lines tested with the **RS**, **CS**, **DS**, and **CD** options on the OPEN "COM... statement. Refer to "OPEN "COM... Statement" in Chapter 4 for details.

If any of the signals that are being tested are turned OFF while the program is executing, I/O statements associated with the communications file won't work. For example, when you execute a PRINT # statement after the CTS or DSR line is turned off, a "Device Fault" (code 25) or "Device Timeout" (code 24) error occurs. The RTS and DTR stay on even if such an error occurs.

You can test for a line disconnect by using the INP function to read the bits in the MODEM Status Register on the Asynchronous Communications Adapter. See the following section, "Testing for Modem Control Signals," for details.

Testing for Modem Control Signals

There are four input control signals picked up by the Asynchronous Communications Adapter. These signals are the CTS and DSR signals described previously, the Carrier Detect (sometimes called Received Line Signal Detect) (pin 8), and Ring Indicator (pin 22). You can specify how you want to test the CTS, DSR, and CD lines with the OPEN "COM... statement. Ring Indicator is not used at all by the communications function in BASIC.

If you need to test for any of these signals in a program, you can check the bits corresponding to these signals in the MODEM Status Register on the Asynchronous Communications Adapter. To read the eight bits in this register, you use the INP function—use INP(&H3FE) to read the register on an unmodified communications adapter, and INP(&H2FE) to read the register on a modified communications adapter. See the "Asynchronous Communications Adapter" section of the *Technical Reference* manual for a description of which bits in the Status Register correspond to which control signals. You can also use the Delta bits in this register to determine if transient signals have appeared on any of the control lines. Note that for a control signal to have meaning, the pin corresponding to that signal must be connected in the cable to your modem or to the other computer.

You can also test for bits in the Line Status Register on the Asynchronous Communications Adapter. Use INP(&H3FD) to access this register on an unmodified communications adapter, and INP(&H2FD) to access it on a modified communications adapter. Again, the bits are described in the *IBM Personal Computer Technical Reference* manual. These bits can be used to determine what types of errors have occurred on receipt of characters from the communications line or whether a break signal has been detected.

Direct Control of Output Control Signals

You can control the RTS or DTR control signals directly from a BASIC program with an OUT statement. The states (ON or OFF) of these signals are controlled by bits in the MODEM Control Register on the Asynchronous Communications Adapter. The address of this register is &H3FC on an unmodified communications adapter and &H2FC on a modified communications adapter. The *IBM Personal Computer Technical Reference* manual describes which of these bits correspond to which signals.

You can also modify bits in the Line Control Register on the Asynchronous Communications Adapter. You should be careful in modifying these bits as most of the bits in this register have been set by BASIC at the time an OPEN statement is executed and changing a bit could cause communications failure. The Line Control Register is at address &H3FB on an unmodified communications adapter and at address &H2FB on a modified communications adapter.

When changing bits in either the MODEM Control Register or the Line Control Register, you should first read the register (with an INP function) and then rewrite the register with only the pertinent bit or bits changed.

A bit you may wish to control in the Line Control Register is bit 6, the Set Break bit. This bit permits you to produce a Break signal on the communications send line. A Break is often used to signal a remote computer to stop transmission. Typically a Break lasts for half a second. To produce such a signal, you must turn ON the Set Break, wait for the desired time of the Break signal, and then turn the bit OFF. The following BASIC statements will produce a Break signal of approximately half a second duration on an unmodified communications adapter.

```
1ØØ IC%=INP(&H3FB) 'get contents of modem register
11Ø IZ%=IC% OR &H4Ø 'turn ON the Set Break bit
11Ø OUT &H3FB,IZ%  'transmit to modem control register
12Ø FOR I=1 TO 5ØØ: NEXT I 'delay half a second
13Ø OUT &H3FB,IC% 'turn Set Break bit OFF in register
```

Communication Errors

Errors occur on communication files in the following order:

1) When opening the file —

    a) "Device Timeout" if one of the signals to be tested (CTS, DSR, or CD) is missing.

2) When reading data —

    a) "Com buffer overflow" if overrun occurs.

    b) "Device I/O error" for overrun, break, parity, or framing errors.

    c) "Device Fault" if you lose DSR or CD.

3) When writing data —

    a) "Device Fault" if you lose CTS, DSR, or CD on a Modem Status Interrupt while BASIC was doing something else.

    b) "Device Timeout" if you lose CTS, DSR, or CD while waiting to put data in the output buffer.

Appendix G. ASCII Character Codes

The following table lists all the ASCII codes (in decimal) and their associated characters. These characters can be displayed using PRINT CHR$(*n*), where *n* is the ASCII code. The column headed "Control Character" lists the standard interpretations of ASCII codes 0 to 31 (usually used for control functions or communications).

Each of these characters may be entered from the keyboard by pressing and holding the Alt key, then pressing the digits for the ASCII code on the numeric keypad. Note, however, that some of the codes have special meaning to the BASIC program editor—the program editor uses its own interpretation for the codes and may not display the special character listed here.

ASCII value	Character	Control character	ASCII value	Character
000	(null)	NUL	032	(space)
001	☺	SOH	033	!
002	☻	STX	034	"
003	♥	ETX	035	#
004	♦	EOT	036	$
005	♣	ENQ	037	%
006	♠	ACK	038	&
007	(beep)	BEL	039	'
008	◘	BS	040	(
009	(tab)	HT	041	)
010	(line feed)	LF	042	*
011	(home)	VT	043	+
012	(form feed)	FF	044	,
013	(carriage return)	CR	045	-
014	♫	SO	046	.
015	☼	SI	047	/
016	►	DLE	048	0
017	◄	DC1	049	1
018	↕	DC2	050	2
019	‼	DC3	051	3
020	¶	DC4	052	4
021	§	NAK	053	5
022	▬	SYN	054	6
023	↨	ETB	055	7
024	↑	CAN	056	8
025	↓	EM	057	9
026	→	SUB	058	:
027	←	ESC	059	;
028	(cursor right)	FS	060	<
029	(cursor left)	GS	061	=
030	(cursor up)	RS	062	>
031	(cursor down)	US	063	?

ASCII value	Character	ASCII value	Character
064	@	096	`
065	A	097	a
066	B	098	b
067	C	099	c
068	D	100	d
069	E	101	e
070	F	102	f
071	G	103	g
072	H	104	h
073	I	105	i
074	J	106	j
075	K	107	k
076	L	108	l
077	M	109	m
078	N	110	n
079	O	111	o
080	P	112	p
081	Q	113	q
082	R	114	r
083	S	115	s
084	T	116	t
085	U	117	u
086	V	118	v
087	W	119	w
088	X	120	x
089	Y	121	y
090	Z	122	z
091	[	123	{
092	\	124	¦
093	]	125	}
094	^	126	~
095	_	127	⌂

ASCII value	Character	ASCII value	Character
128	Ç	160	á
129	ü	161	í
130	é	162	ó
131	â	163	ú
132	ä	164	ñ
133	à	165	Ñ
134	å	166	ª
135	ç	167	º
136	ê	168	¿
137	ë	169	⌐
138	è	170	¬
139	ï	171	½
140	î	172	¼
141	ì	173	¡
142	Ä	174	«
143	Å	175	»
144	É	176	░
145	æ	177	▒
146	Æ	178	▓
147	ô	179	│
148	ö	180	┤
149	ò	181	╡
150	û	182	╢
151	ù	183	╖
152	ÿ	184	╕
153	Ö	185	╣
154	Ü	186	║
155	¢	187	╗
156	£	188	╝
157	¥	189	╜
158	Pt	190	╛
159	ƒ	191	┐

ASCII value	Character	ASCII value	Character
192	└	224	α
193	┴	225	β
194	┬	226	Γ
195	├	227	π
196	─	228	Σ
197	┼	229	σ
198	╞	230	μ
199	╟	231	τ
200	╚	232	Φ
201	╔	233	Θ
202	╩	234	Ω
203	╦	235	δ
204	╠	236	∞
205	═	237	Ø
206	╬	238	ϵ
207	╧	239	$\cap$
208	╨	240	$\equiv$
209	╤	241	$\pm$
210	╥	242	$\geq$
211	╙	243	$\leq$
212	╘	244	⌠
213	╒	245	⌡
214	╓	246	$\div$
215	╫	247	$\approx$
216	╪	248	°
217	┘	249	•
218	┌	250	·
219	█	251	$\surd$
220	▄	252	^{n}
221	▌	253	^{2}
222	▐	254	■
223	▀	255	(blank 'FF')

Extended Codes

For certain keys or key combinations that cannot be represented in standard ASCII code, an extended code is returned by the INKEY$ system variable. A null character (ASCII code 000) will be returned as the first character of a two-character string. If a two-character string is received by INKEY$, then you should go back and examine the second character to determine the actual key pressed. Usually, but not always, this second code is the scan code of the primary key that was pressed. The ASCII codes (in decimal) for this second character, and the associated key(s) are listed on the following page.

Second Code	Meaning
3	(null character) NUL
15	(shift tab) |◄—
16-25	Alt- Q, W, E, R, T, Y, U, I, O, P
30-38	Alt- A, S, D, F, G, H, J, K, L
44-50	Alt- Z, X, C, V, B, N, M
59-68	function keys F1 through F10 (when disabled as soft keys)
71	Home
72	Cursor Up
73	Pg Up
75	Cursor Left
77	Cursor Right
79	End
80	Cursor Down
81	Pg Dn
82	Ins
83	Del
84-93	F11-F20 (Shift- F1 through F10)
94-103	F21-F30 (Ctrl- F1 through F10)
104-113	F31-F40 (Alt- F1 through F10)
114	Ctrl-PrtSc
115	Ctrl-Cursor Left (Previous Word)
116	Ctrl-Cursor Right (Next Word)
117	Ctrl-End
118	Ctrl-Pg Dn
119	Ctrl-Home
120-131	Alt- 1,2,3,4,5,6,7,8,9,0,-,=
132	Ctrl-Pg Up

NOTES

Appendix H. Hexadecimal Conversion Table

Hex	Decimal	Hex	Decimal
1	1	10	16
2	2	20	32
3	3	30	48
4	4	40	64
5	5	50	80
6	6	60	96
7	7	70	112
8	8	80	128
9	9	90	144
A	10	A0	160
B	11	B0	176
C	12	C0	192
D	13	D0	208
E	14	E0	224
F	15	F0	240
100	256	1000	4096
200	512	2000	8192
300	768	3000	12288
400	1024	4000	16384
500	1280	5000	20480
600	1536	6000	24576
700	1792	7000	28672
800	2048	8000	32768
900	2304	9000	36864
A00	2560	A000	40960
B00	2816	B000	45056
C00	3072	C000	49152
D00	3328	D000	53248
E00	3584	E000	57344
F00	3840	F000	61440

NOTES

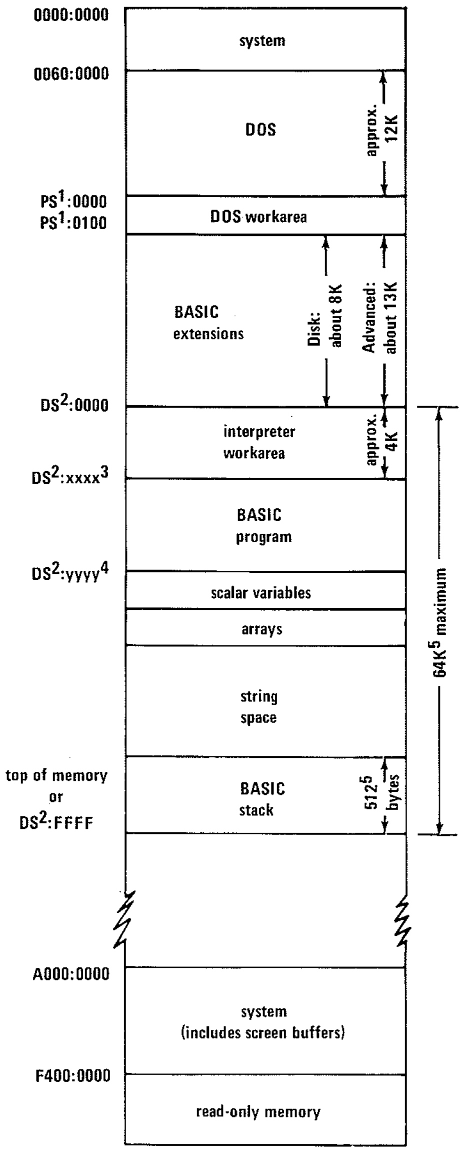
Appendix I. Technical Information and Tips

This appendix contains more specific technical information pertaining to BASIC. Included are a memory map, descriptions of how BASIC stores data internally, and some special techniques you can use to improve program performance.

Other information may be found in the *IBM Personal Computer Technical Reference* manual.

Memory Map

The following is a memory map for Disk and Advanced BASIC. DOS and the BASIC extensions are not present for Cassette BASIC. Addresses are in hexadecimal in the form *segment:offset*.

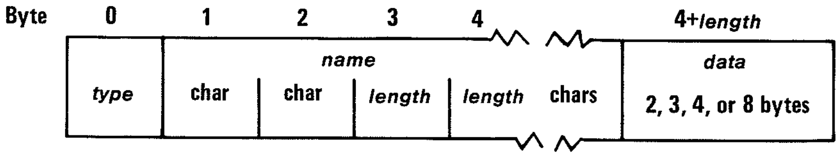


Notes:

1. PS refers to DOS Program Segment
2. DS refers to BASIC's Data Segment
3. the number xxxx is in locations DS:30 and DS:31 (low byte, high byte)
4. the number yyyy is in locations DS:358, DS:359 (low byte, high byte)
5. or set by CLEAR command

How Variables Are Stored

Scalar variables are stored in BASIC's data area as follows:



type identifies the variable's type:

2	integer
3	string
4	single-precision
8	double-precision

name is the name of the variable. The first two characters of the name are stored in the bytes 1 and 2. Byte 3 tells how many more characters are in the variable name. These additional characters are stored starting at byte 4.

Note that this means any variable name will take up at least three bytes. A one- or two-character name will occupy exactly three bytes; an x character name will occupy x+1 bytes.

data follows the name of the variable, and may be either two, three, four, or eight bytes long (as described by *type*). The value returned by the VARPTR function points to this data.

For string variables, *data* is the *string descriptor:*

- The first byte of the string descriptor contains the length of the string (0 to 255).
- The last two bytes of the string descriptor contain the address of the string in BASIC's data space (the offset into the default segment). Addresses are stored with the low byte first and the high byte second, so:
  - The second byte of the string descriptor contains the low byte of the offset.
  - The third byte of the string descriptor contains the high byte of the offset.

For numeric variables *data* contains the actual value of the variable:

- Integer values are stored in two bytes, with the low byte first and the high byte second.
- Single-precision values are stored in four bytes in BASIC's internal floating point binary format.
- Double-precision values are stored in eight bytes in BASIC's internal floating point binary format.

BASIC File Control Block

When you call VARPTR with a file number as an argument, the returned value is the address of the BASIC file control block. The address is specified as an offset into BASICs Data Segment. (Note that the BASIC file control block is not the same as the DOS file control block.)

Information contained in the file control block is as follows (offsets are relative to the value returned by VARPTR):

Offset	Length	Description
0	1	The mode in which the file was opened: 1 - Input only 2 - Output only 4. - Random 16 - Append only 32 - Internal use 128 - Internal use
1	38	DOS file control block
39	2	For sequential files, the number of sectors read or written. For random files, contains 1 + the last record number read or written.
41	1	Number of bytes in sector when read or written.
42	1	Number of bytes left in input buffer.
43	3	(reserved)
46	1	Device number: 0,1 - Diskette drives A: and B: 248 - LPT3: 249 - LPT2: 250 - COM2: 251 - COM1: 252 - CAS1: 253 - LPT1: 254 - SCRN: 255 - KYBD:

Offset	Length	Description
47	1	Device width.
48	1	Position in buffer for PRINT #.
49	1	Internal use during LOAD and SAVE. Not used for data files.
50	1	Output position used during tab expansion.
51	128	Physical data buffer. Used to transfer data between DOS and BASIC. Use this offset to examine data in sequential I/O mode.
179	2	Variable length record size. Default is 128. Set by length parameter on OPEN statement.
181	2	Current physical record number.
183	2	Current logical record number.
185	1	(reserved)
186	2	Diskette files only. Position for PRINT #, INPUT #, and WRITE #.
188	n	Actual FIELD data buffer. Size n is determined by the **/S:** option on the BASIC command. Use this offset to examine file data in random mode.

Keyboard Buffer

Characters typed on the keyboard are saved in the keyboard buffer until they are processed. Up to 15 characters can be held in the buffer; if you try to type more than 15 characters, the computer beeps.

INKEY$ will read only one character from the keyboard buffer even if there are several characters pending there. INPUT$ can be used to read multiple characters; however, if the requested number of characters are not already present in the buffer, BASIC will wait until enough characters are typed.

The system keyboard buffer may be cleared by the following lines of code:

```
DEF SEG=Ø: POKE 1Ø5Ø, PEEK(1Ø52)
```

This technique could be useful, for example, to clear the buffer before you ask the user to "press any key."

BASIC has its own line buffer, where the program editor acts on characters that are received from the system keyboard buffer. BASIC's line buffer may be cleared using the following code:

```
DEF SEG: POKE 1Ø6,Ø
```

Search Order for Adapters

The printers associated with LPT1:, LPT2:, and LPT3: are assigned when you switch your computer on. The system looks for printer adapters in a particular sequence; the first printer adapter found becomes LPT1:, the second adapter (if one exists) becomes LPT2:, and the third (if it exists) becomes LPT3:. The search order is as follows:

1. An IBM Monochrome Display and Parallel Printer Adapter
2. A Parallel Printer Adapter
3. A Parallel Printer Adapter which has been modified to change its base address

If a printer was re-routed using the MODE command from DOS, the change is effective in BASIC as well.

The communication devices COM1: and COM2: are assigned in a manner similar to the printers. Their search order is:

1. An Asynchronous Communications Adapter
2. A modified Asynchronous Communications Adapter

Switching Displays

If you have both the Color/Graphics Monitor Adapter and the IBM Monochrome Display and Parallel Printer Adapter in your IBM Personal Computer, the one BASIC will normally write to would be the Monochrome Display. However, you can switch from one display to the other from BASIC by using the following code:

```
10 ' switch to monochrome adapter
20 DEF SEG = 0
30 POKE &H410, (PEEK(&H410) OR &H30)
40 SCREEN 0
50 WIDTH 40
60 WIDTH 80
70 LOCATE ,,1,12,13
```

```
10 ' switch to color adapter
20 DEF SEG = 0
30 POKE &H410, (PEEK(&H410) AND &HCF) OR &H10
40 SCREEN 1,0,0,0
50 SCREEN 0
60 WIDTH 40
70 LOCATE ,,1,6,7
```

Note: When you use this technique, the screen you are switching *to* is cleared. Also, you may need to keep track of the cursor location independently for each display.

Some Techniques with Color

Sixteen Background Colors: In text mode, if you are willing to give up blink, you can get all 16 colors (0-15) for the background color. Do the following:

In 40-column width: `OUT &H3D8,8`

In 80-column width: `OUT &H3D8,9`

Character Color in Graphics Mode: You can display regular text characters while in graphics mode. In medium resolution, the foreground color of the characters is color number 3; the background is color number 0.

You can change the foreground color of the characters from 3 to 2 or 1 by performing a:

```
DEF SEG: POKE &H4E, color
```

where *color* is the desired foreground color (1, 2, or 3— 0 is *not* allowed). Later PRINTs will use the specified foreground color.

Tips and Techniques

Often there are several different ways you can code something in BASIC and still get the same function. This section contains some general hints for coding to improve program performance.

GENERAL

- **Combine statements** where convenient to take advantage of the 255 character statement length. For example:

  Do

  ```
  100 FOR I=1 TO 10: READ A(I): NEXT I
  ```

  Instead of

  ```
  100 FOR I=1 TO 10
  110 READ A(I)
  120 NEXT I
  ```

- **Avoid repetitive evaluation of expressions.** If you do the identical calculation in several statements, you can evaluate the expression once and save the result in a variable for use in later statements. For example:

  ```
  Do                    Instead of

  300 X=C*3+D           310 A=C*3+D+Y
  310 A=X+Y             320 B=C*3+D+Z
  320 B=X+Z
  ```

  However, assigning a constant to a variable is faster than assigning the value of another variable to the variable.

- **Use simple arithmetic.** In general, addition is performed faster than multiplication, and multiplication is faster than division or exponentiation.

  Consider these example:

```
Do                   Instead of

25Ø  B=A*.5          25Ø  B=A/2
5ØØ  B=A+A           5ØØ  B=A*2
65Ø  B=A*A*A*        65Ø  B=A^3
75Ø  B%=A%\4         75Ø  B%=INT(A%/4)
```

- **Use built-in functions.** Use the built-in system functions where possible; they always execute faster than the same capability written in BASIC.

- **Use remarks sparingly.** It takes a small amount of time for BASIC to identify a remark. Use the single quote (') to place remarks at the end of the line rather than using a separate statement for them when possible. This improves performance and saves storage by eliminating the need for a line number. For example:

  Do

```
1Ø  FOR I=1 TO 1Ø
2Ø  A(I)=3Ø  ' initialize A
3Ø  NEXT I
```

  Instead of

```
1Ø  FOR I=1 TO 1Ø
15  ' initialize A
2Ø  A(I)=3Ø
3Ø  NEXT I
```

- Just a note about IBM Personal Computer BASIC— When BASIC wants to branch to a particular line number, it doesn't know exactly where in memory that line is. Therefore BASIC has to search through the line numbers in the program, starting at the beginning, to find the line it's looking for.

In some other BASICs, this search must be performed each time the branch occurs in the program. In IBM Personal Computer BASIC, the search is only performed once, and thereafter the branch is direct. So placing frequently-used subroutines at the beginning of the program will not make your program run faster.

LOGIC CONTROL

- **Use the capabilities of the IF statement.** By using AND and OR and the ELSE clause, you can often avoid the need for more IF statements and additional code in the program. For example:

  Do

  ```
  200 IF A=B AND C=D THEN Z=12 ELSE Z=B
  ```

  Instead of

  ```
  200 IF A=B THEN GOTO 210
  205 GOTO 215
  210 IF C=D THEN 225
  215 Z=B
  220 GOTO 230
  225 Z=12
  230 ...
  ```

- **Order IF statements** so the most frequently occurring condition is tested first. This avoids having to make extra tests. For example, suppose you have a data entry file for customer orders which consists of different record types and numerous individual transactions.

A typical record group looks like this:

Type code	Record type
A	Header
B	Customer name and address
C	Transaction
C	Transaction
.	.
.	.
C	Transaction
D	Trailer

Do

```
100 IF TYPE$="C" THEN 3000
110 IF TYPE$="A" THEN 1000
120 IF TYPE$="B" THEN 2000
130 IF TYPE$="D" THEN 4000
```

Instead of

```
100 IF TYPE$="A" THEN 1000
110 IF TYPE$="B" THEN 2000
120 IF TYPE$="C" THEN 3000
130 IF TYPE$="D" THEN 4000
```

If you had 100 groups, with 10 transactions per group, moving the test to the beginning of the list results in 1800 fewer IF statements being executed.

Another example of ordering IF statements in a cascade so less tests need to be performed:

DO

```
200 IF A<>1 THEN 250
210 IF B=1 THEN X=0
220 IF B=2 THEN X=1
230 IF B=3 THEN X=2
240 GOTO 280
250 IF B=1 THEN X=3
260 IF B=2 THEN X=4
270 IF B=3 THEN X=5
280 ...
```

Instead of

```
200 IF A=1 AND B=1 THEN X=0
210 IF A=1 AND B=2 THEN X=1
220 IF A=1 AND B=3 THEN X=2
230 IF A<>1 AND B=1 THEN X=3
240 IF A<>1 AND B=2 THEN X=4
250 IF A<>1 AND B=3 THEN X=5
```

LOOPS

- **Use integer counters** on FOR...NEXT loops when possible. Integer arithmetic is performed faster than single- and double-precision arithmetic.

- **Omit the variable on the NEXT statement** where possible. If you include the variable, BASIC takes a little time to check to see that it is correct. It may be necessary to include the variable on the NEXT statement if you are branching out of nested loops. Refer to "FOR and NEXT Statements" in Chapter 4 for more information.

- **Use FOR...NEXT loops** instead of using the IF, GOTO combination of statements.

For example:

```
Do                       Instead of

200 FOR I=1 TO 10        200 I=1
  .                      210 ...
  .                        .
  .                        .
  .                      290 I=I+1
300 NEXT I               300 IF I<=10 THEN 210
```

- **Remove unnecessary code from loops.** This includes statements which don't affect the loop, as well as non-executable statements such as REM and DATA. For example:

Do

```
  10 A=B+1
  20 FOR X=1 TO 100
  30 IF D(X)>A THEN D(X)=A
  40 NEXT X
```

Instead of

```
  10 FOR X=1 TO 100
  20 A=B+1
  30 IF D(X)>A THEN D(X)=A
  40 NEXT X
```

In the preceding example, it is not necessary to calculate the value of A each time through the loop, because the loop never changes the value of A.

The next example shows a non-executable statement.

```
Do                          Instead of

200 DATA 5, 12, 1943        200 FOR I=1 TO 100
210 FOR I=1 TO 100          210 DATA 5, 12, 1943
     .                           .
     .                           .
     .                           .
300 NEXT I                  300 NEXT I
```

Refer also to "Performance Hints" in Appendix B for some tips relating to diskette files.

Appendix J. Glossary

This part of the book explains many of the technical terms you may run across while programming in BASIC.

absolute coordinate form: In graphics, specifying the location of a point with respect to the origin of the coordinate system.

access mode: A technique used to obtain a specific logical record from, or put a logical record into, a file.

accuracy: The quality of being free from error. On a machine this is actually measured, and refers to the size of the error between the actual number and its value as stored in the machine.

active page: On the Color/Graphics Monitor Adapter, the screen buffer which has information written to it. It may be different from the screen buffer whose information is being displayed.

adapter: A mechanism for attaching parts.

address: The location of a register, a particular part of memory, or some other data source or destination. Or, to refer to a device or a data item by its address.

addressable point: In computer graphics, any point in a display space that can be addressed. Such points are finite in number and form a discrete grid over the display space.

algorithm: A finite set of well-defined rules for the solution of a problem in a finite number of steps.

allocate: To assign a resource, such as a diskette file or a part of memory, to a specific task.

alphabetic character: A letter of the alphabet.

alphameric or alphanumeric: Pertaining to a character set that contains letters and digits.

application program: A program written by or for a user which applies to the user's work. For example, a payroll application program.

argument: A value that is passed from a calling program to a function.

arithmetic overflow: Same as overflow.

array: An arrangement of elements in one or more dimensions.

ASCII: American National Standard Code for Information Interchange. The standard code used for exchanging information among data processing systems and associated equipment. An ASCII file is a text file where the characters are represented in ASCII codes.

asynchronous: Without regular time relationship; unpredictable with respect to the execution of a program's instructions.

attribute: A property or characteristic of one or more items.

background: The area which surrounds the subject. In particular, the part of the display screen surrounding a character.

backup: Pertaining to a system, device, file, or facility that can be used in case of a malfunction or loss of data.

baud: A unit of signalling speed equal to the number of discrete conditions or signal events per second.

binary: Pertaining to a condition that has two possible values or states. Also, refers to the Base 2 numbering system.

bit: A binary digit.

blank: A part of a data medium in which no characters are recorded. Also, the space character.

blinking: An intentional regular change in the intensity of a character on the screen.

boolean value: A numeric value that is interpreted as "true" (if it is not zero) or "false" (if it is zero).

bootstrap: An existing version, perhaps a primitive version, of a computer program that is used to establish another version of the program. Can be thought of as a program which loads itself.

bps: Bits per second.

bubble sort: A technique for sorting a list of items into sequence. Pairs of items are examined, and exchanged if they are out of sequence. This process is repeated until the list is sorted.

buffer: An area of storage which is used to compensate for a difference in rate of flow of data, or time of occurrence of events, when transferring data from one device to another. Usually refers to an area reserved for I/O operations, into which data is read or from which data is written.

bug: An error in a program.

byte: The representation of a character in binary. Eight bits.

call: To bring a computer program, a routine, or a subroutine into effect, usually by specifying the entry conditions and jumping to an entry point.

carriage return character (CR): A character that causes the print or display position to move to the first position on the same line.

channel: A path along which signals can be sent, for example, a data channel or an output channel.

character: A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A connected sequence of characters is called a character *string*.

clock: A device that generates periodic signals used for synchronization. Each signal is called a clock pulse or clock tick.

code: To represent data or a computer program in a symbolic form that can be accepted by a computer; to write a routine. Also, loosely, one or more computer programs, or part of a program.

comment: A statement used to document a program. Comments include information that may be helpful in running the program or reviewing the output listing.

communication: The transmission and reception of data.

complement: An "opposite." In particular, a number that can be derived from a given number by subtracting it from another given number.

compression: Arranging data so it takes up a minimal amount of space.

concatenation: The operation that joins two strings together in the order specified, forming a single string with a length equal to the sum of the lengths of the two strings.

constant: A fixed value or data item.

control character: A character whose occurrence in a particular context initiates, modifies, or stops a control operation. A control operation is an action that affects the recording, processing, transmission, or interpretation of data; for example, carriage return, font change, or end of transmission.

coordinates: Numbers which identify a location on the display.

cursor: A movable marker that is used to indicate a position on the display.

debug: To find and eliminate mistakes in a program.

default: A value or option that is assumed when none is specified.

delimiter: A character that groups or separates words or values in a line of input.

diagnostic: Pertaining to the detection and isolation of a malfunction or mistake.

directory: A table of identifiers and references to the corresponding items of data. For example, the directory for a diskette contains the names of files on the diskette (identifiers), along with information that tells DOS where to find the file on the diskette.

disabled: A state that prevents the occurrence of certain types of interruptions.

DOS: Disk Operating System. In this book, refers only to the IBM Personal Computer Disk Operating System.

dummy: Having the appearance of a specified thing but not having the capacity to function as such. For example, a dummy argument to a function.

duplex: In data communication, pertaining to a simultaneous two-way independent transmission in both directions. Same as full duplex.

dynamic: Occurring at the time of execution.

echo: To reflect received data to the sender. For example, keys pressed on the keyboard are usually echoed as characters displayed on the screen.

edit: To enter, modify, or delete data.

element: A member of a set; in particular, an item in an array.

enabled: A state of the processing unit that allows certain types of interruptions.

end of file (EOF): A "marker" immediately following the last record of a file, signalling the end of that file.

event: An occurrence or happening; in IBM Personal Computer Advanced BASIC, refers particularly to the events tested by the COM(n), KEY(n), PEN, and STRIG(n).

execute: To perform an instruction or a computer program.

extent: A continuous space on a diskette, occupied or reserved for a particular file.

fault: An accidental condition that causes a device to fail to perform in a required manner.

field: In a record, a specific area used for a particular category of data.

file: A set of related records treated as a unit.

fixed-length: Referring to something in which the length does not change. For example, random files have fixed-length records; that is, each record has the same length as all the other records in the file.

flag: Any of various types of indicators used for identification, for example, a character that signals the occurrence of some condition.

floppy disk: A diskette.

folding: A technique for converting data to a desired form when it doesn't start out in that form. For example, lowercase letters may be folded to uppercase.

font: A family or assortment of characters of a particular size and style.

foreground: The part of the display area that is the character itself.

format: The particular arrangement or layout of data on a data medium, such as the screen or a diskette.

form feed (FF): A character that causes the print or display position to move to the next page.

function: A procedure which returns a value depending on the value of one or more independent variables in a specified way. More generally, the specific purpose of a thing, or its characteristic action.

function key: One of the ten keys labeled F1 through F10 on the left side of the keyboard.

garbage collection: Synonym for housecleaning.

graphic: A symbol produced by a process such as handwriting, printing, or drawing.

half duplex: In data communication, pertaining to an alternate, one way at a time, independent transmission.

hard copy: A printed copy of machine output in a visually readable form.

header record: A record containing common, constant, or identifying information for a group of records that follows.

hertz (Hz): A unit of frequency equal to one cycle per second.

hierarchy: A structure having several levels, arranged in a tree-like form. "Hierarchy of operations" refers to the relative priority assigned to arithmetic or logical operations which must be performed.

host: The primary or controlling computer in a multiple computer installation.

housecleaning: When BASIC compresses string space by collecting all of its useful data and frees up unused areas of memory that were once used for strings.

implicit declaration: The establishment of a dimension for an array without it having been explicitly declared in a DIM statement.

increment: A value used to alter a counter.

initialize: To set counters, switches, addresses, or contents of memory to zero or other starting values at the beginning of, or at prescribed points in, the operation of a computer routine.

instruction: In a programming language, any meaningful expression that specifies one operation and its operands, if any.

integer: One of the numbers 0, ±1, ±2, ±3, ...

integrity: Preservation of data for its intended purpose; data integrity exists as long as accidental or malicious destruction, alteration, or loss of data are prevented.

interface: A shared boundary.

interpret: To translate and execute each source language statement of a computer program before translating and executing the next statement.

interrupt: To stop a process in such a way that it can be resumed.

invoke: To activate a procedure at one of its entry points.

joystick: A lever that can pivot in all directions and is used as a locator device.

justify: To align characters horizontally or vertically to fit the positioning constraints of a required format.

K: When referring to memory capacity, two to the tenth power or 1024 in decimal notation.

keyword: One of the predefined words of a programming language; a reserved word.

leading: The first part of something. For example, you might refer to leading zeroes or leading blanks in a character string.

light pen: A light sensitive device that is used to select a location on the display by pointing it at the screen.

line: When referring to text on a screen or printer, one or more characters output before a return to the first print or display position. When referring to input, a string of characters accepted by the system as a single block of input; for example, all characters entered before you press the Enter key. In graphics, a series of points drawn on the screen to form a straight line. In data communications, any physical medium, such as a wire or microwave beam, that is used to transmit data.

line feed (LF): A character that causes the print or display position to move to the corresponding position on the next line.

literal: An explicit representation of a value, especially a string value; a constant.

location: Any place in which data may be stored.

loop: A set of instructions that may be executed repeatedly while a certain condition is true.

M: Mega; one million. When referring to memory, two to the twentieth power; 1,048,576 in decimal notation.

machine infinity: The largest number that can be represented in a computer's internal format.

mantissa: For a number expressed in floating point notation, the numeral that is not the exponent.

mask: A pattern of characters that is used to control the retention or elimination of another pattern of characters.

matrix: An array with two or more dimensions.

matrix printer: A printer in which each character is represented by a pattern of dots.

menu: A list of available operations. You select which operation you want from the list.

minifloppy: A 5-1/4 inch diskette.

nest: To incorporate a structure of some kind into another structure of the same kind. For example, you can nest loops within other loops, or call subroutines from other subroutines.

notation: A set of symbols, and the rules for their use, for the representation of data.

null: Empty, having no meaning. In particular, a string with no characters in it.

octal: Pertaining to a Base 8 number system.

offset: The number of units from a starting point (in a record, control block, or memory) to some other point. For example, in BASIC the actual address of a memory location is given as an offset in bytes from the location defined by the DEF SEG statement.

on-condition: An occurrence that could cause a program interruption. It may be the detection of an unexpected error, or of an occurrence that is expected, but at an unpredictable time.

operand: That which is operated upon.

operating system: Software that controls the execution of programs; often used to refer to DOS.

operation: A well-defined action that, when applied to any permissible combination of known entities, produces a new entity.

overflow: When the result of an operation exceeds the capacity of the intended unit of storage.

overlay: To use the same areas of memory for different parts of a computer program at different times.

overwrite: To record into an area of storage so as to destroy the data that was previously stored there.

pad: To fill a block with dummy data, usually zeros or blanks.

page: Part of the screen buffer that can be displayed and/or written on independently.

parameter: A name in a procedure that is used to refer to an argument passed to that procedure.

parity check: A technique for testing transmitted data. Typically, a binary digit is appended to a group of binary digits to make the sum of all the digits either always even (even parity) or always odd (odd party).

pixel: A graphics "point." Also, the bits which contain the information for that point.

port: An access point for data entry or exit.

position: In a string, each location that may be occupied by a character and that may be identified by a number.

precision: A measure of the ability to distinguish between nearly equal values.

prompt: A question the computer asks when it needs you to supply information.

protect: To restrict access to or use of all, or part of, a data processing system.

queue: A line or list of items waiting for service; the first item that went in the queue is the first item to be serviced.

random access memory: Storage in which you can read and write to any desired location. Sometimes called direct access storage.

range: The set of values that a quantity or function may take.

raster scan: A technique of generating a display image by a line-by-line sweep across the entire display screen. This is the way pictures are created on a television screen.

read-only: A type of access to data that allows it to be read but not modified.

record: A collection of related information, treated as a unit. For example, in stock control, each invoice might be one record.

recursive: Pertaining to a process in which each step makes use of the results of earlier steps, such as when a function calls itself.

relative coordinates: In graphics, values that identify the location of a point by specifying displacements from some other point.

reserved word: A word that is defined in BASIC for a special purpose, and that you cannot use as a variable name.

resolution: In computer graphics, a measure of the sharpness of an image, expressed as the number of lines per unit of length discernible in that area.

routine: Part of a program, or a sequence of instructions called by a program, that may have some general or frequent use.

row: A horizontal arrangement of characters or other expressions.

scalar: A value or variable that is not an array.

scale: To change the representation of a quantity, expressing it in other units, so that its range is brought within a specified range.

scan: To examine sequentially, part by part. See raster scan.

scroll: To move all or part of the display image vertically or horizontally so that new data appears at one edge as old data disappears at the opposite edge.

segment: A particular 64K-byte area of memory.

sequential access: An access mode in which records are retrieved in the same order in which they were written. Each successive access to the file refers to the next record in the file.

stack: A method of temporarily storing data so that the last item stored is the first item to be processed.

statement: A meaningful expression that may describe or specify operations and is complete in the context of the BASIC programming language.

stop bit: A signal following a character or block that prepares the receiving device to receive the next character or block.

storage: A device, or part of a device, that can retain data. Memory.

string: A sequence of characters.

subscript: A number that identifies the position of an element in an array.

syntax: The rules governing the structure of a language.

table: An arrangement of data in rows and columns.

target: In an assignment statement, the variable whose value is being set.

telecommunication: Synonym for data communication.

terminal: A device, usually equipped with a keyboard and display, capable of sending and receiving information.

toggle: Pertaining to anything having two stable states; to switch back and forth between the two states.

trailing: Located at the end of a string or number. For example, the number 1000 has three trailing zeros.

trap: A set of conditions that describe an event to be intercepted and the action to be taken after the interception.

truncate: To remove the ending elements from a string.

two's complement: A form for representing negative numbers in the binary number system.

typematic key: A key that repeats as long as you hold it down.

update: To modify, usually a master file, with current information.

variable: A quantity that can assume any of a given set of values.

variable-length record: A record having a length independent of the length of other records in the file.

vector: In graphics, a directed line segment. More generally, an ordered set of numbers, and so, a one-dimensional array.

wraparound: The technique for displaying items whose coordinates lie outside the display area.

write: To record data in a storage device or on a data medium.

INDEX

IBM

The Personal Computer
Software Library

Product Comment Form

BASIC 6025013

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the Authorized IBM Personal Computer Dealer in your area.

Comments:

If you wish a reply, provide your name and address in this space.

Name ____________________

Address ____________________

City ____________ State ____________

Zip Code ____________

Tape Please do not staple Tape

Fold here

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

IBM

The Personal Computer
Software Library

Product Comment Form

BASIC 6025013

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the Authorized IBM Personal Computer Dealer in your area.

Comments:

If you wish a reply, provide your name and address in this space.

Name ______________________________

Address ______________________________

City ______________ State ______________

Zip Code ___________

Tape

Please do not staple

Tape

Fold here

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

IBM

The Personal Computer
Software Library

Product Comment Form

BASIC 6025013

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the Authorized IBM Personal Computer Dealer in your area.

Comments:

If you wish a reply, provide your name and address in this space.

Name ______________________________

Address ______________________________

City ______________ State ______________

Zip Code ______________

Tape Please do not staple Tape

Fold here

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

The Personal Computer
Software Library

Product Comment Form

BASIC 6025013

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the Authorized IBM Personal Computer Dealer in your area.

Comments:

If you wish a reply, provide your name and address in this space.

Name ______________________________

Address ______________________________

City ______________ State ______________

Zip Code ______________

Tape Please do not staple Tape

Fold here

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

The Personal Computer
Software Library

Product Comment Form

BASIC 6025013

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the Authorized IBM Personal Computer Dealer in your area.

Comments:

If you wish a reply, provide your name and address in this space.

Name ______________________________

Address ______________________________

City ______________ State ______________

Zip Code ______________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Tape Please do not staple Tape