

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

APPENDIX A. MESSAGES	A-5
APPENDIX B. BASIC DISKETTE INPUT AND OUTPUT	B-1
Specifying Filenames	B-2
Commands for Program Files	B-2
Protected Files	B-3
Diskette Data Files - Sequential and Random I/O	B-4
Sequential Files	B-4
Creating and Accessing a Sequential File	B-4
Adding Data to a Sequential File ...	B-7
Random Files	B-8
Creating a Random File	B-9
Accessing a Random File	B-10
An Example Program	B-12
Performance Hints	B-15
APPENDIX C. MACHINE LANGUAGE SUBROUTINES	C-1
Setting Memory Aside for Your Subroutines	C-2
Getting the Subroutine Code into Memory	C-3
Poking a Subroutine into Memory ...	C-4
Loading the Subroutine from a File ...	C-5
Calling the Subroutine from Your BASIC Program	C-8
Common Features of CALL and USR ...	C-8
CALL Statement	C-10
USR Function Calls	C-14

**APPENDIX D. CONVERTING PROGRAMS TO
IBM PERSONAL COMPUTER BASIC ... D-1**

File I/O	D-1
Graphics	D-1
IF... THEN	D-2
Line Feeds	D-3
Logical Operations	D-3
MAT Functions	D-4
Multiple Assignments	D-4
Multiple Statements	D-4
PEEKs and POKEs	D-4
Relational Expressions	D-5
Remarks	D-5
Rounding of Numbers	D-5
Sounding the Bell	D-5
String Handling	D-6
Use of Blanks	D-7
Other	D-7

**APPENDIX E. MATHEMATICAL
FUNCTIONS E-1**

APPENDIX F. COMMUNICATIONS F-1

Opening a Communications File	F-1
Communication I/O	F-1
GET and PUT for Communications Files	F-2
I/O Functions	F-2
INPUT\$ Functions	F-3
An Example Program	F-4
Notes on the Program	F-5

Operation of Control Signals	F-6
Control of Output Signals with OPEN ..	F-6
Use of Input Control Signals	F-7
Testing for Modem Control Signals ...	F-7
Direct Control of Output Control Signals	F-8
Communication Errors	F-10

APPENDIX G. ASCII CHARACTER CODES	G-1
Extended Codes	G-6
APPENDIX H. HEXADECIMAL CONVERSION TABLE	H-1
APPENDIX I. TECHNICAL INFORMATION AND TIPS	I-1
Memory Map	I-2
How Variables Are Stored	I-3
BASIC File Control Block	I-4
Keyboard Buffer	I-7
Search Order for Adapters	I-7
Switching Displays	I-8
Some Techniques with Color	I-9
Tips and Techniques	I-10
APPENDIX J. GLOSSARY	J-1

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.

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

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.

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.

- 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.

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.

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).

- 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.

- 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.

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 "0",#1,"DATA" 'step 1
200 WRITE #1,A$,B$,C$ 'step 2
300 CLOSE #1 'step 3
400 OPEN "1",#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";NS
25 IF NS="DONE" THEN CLOSE: END
30 INPUT "DEPARTMENT";DS
40 INPUT "DATE HIRED";HS
50 WRITE #1,NS,DS,HS
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? EBENEZER 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,NS,DS,HS
30 IF RIGHTS(HS,2)="78" THEN PRINT NS
40 GOTO 20
RUN
```

```
EBENEZER 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$=MK$$(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$=MK$$ (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
1Ø OPEN 'ABC' FOR INPUT AS #1
2Ø OPEN 'DEF' FOR OUTPUT AS #2
3Ø IF EOF(1) THEN CLOSE: END
4Ø INPUT #1,A$
5Ø PRINT #2,A$
6Ø GOTO 3Ø
7Ø 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
1Ø OPEN 'PQR' AS #1 LEN=512
15 ON ERROR GOTO 9Ø
2Ø FOR I=1 TO 3ØØØ
3Ø PRINT #1,'MELH'
4Ø NEXT
45 PRINT #1,'/eof'
5Ø ON ERROR GOTO Ø: PUT #1: CLOSE
6Ø END
9Ø 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
1Ø OPEN 'PQR' AS #1 LEN=512
2Ø OPEN 'XYZ' AS #2 LEN=512
3Ø ON ERROR GOTO 8Ø
4Ø GET #1
5Ø INPUT #1,A$
6Ø PRINT #2,A$
7Ø IF A$<>' /eof' THEN 5Ø ELSE
    ON ERROR GOTO Ø: PUT #2: CLOSE: END
8Ø IF ERL=5Ø 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=&H17000
```

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
10 DEFINT A-Z
20 DEF SEG=&H1700
30 FOR I=0 TO 21
40 READ J
50 POKE I,J
60 NEXT
70 SUBRT=0
80 A=2:B=3:C=0
90 CALL SUBRT(A,B,C)
100 PRINT C
110 END
120 DATA &H55,&H8B,&HEC,&H8B,&H76,&H0A
130 DATA &H5B,&H04,&H8B,&H76,&H08
140 DATA &H03,&H04,&H8B,&H7E,&H06
150 DATA &H09,&H05,&H5D,&HCA,&H06,&H00
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:

$$\text{offset from BP} = 2*(n-m)+6$$

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.