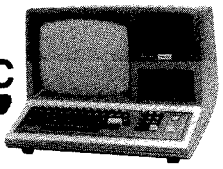


Introduction

Start-Up

Under TRSDOS READY, type:

```
BASIC (ENTER)
```

TRSDOS will load BASIC and begin the "initialization dialog."

If you want to *recover* a Disk BASIC program after returning to TRSDOS for a DIR or other TRSDOS command, use this command under TRSDOS READY:

```
BASIC * (ENTER)
```

You will go directly to BASIC's READY mode without any initialization dialog. If you had a program in memory, it should still be there. You may not be able to run the program. To be safe, you should immediately save the program, go to TRSDOS, then start BASIC again (no asterisk).

Note: If you have overlaid user memory while in TRSDOS, your program will be erased. In such a case, you should not restart BASIC, but should use the normal BASIC start-up procedure.

Initialization

When you start Disk BASIC, you are first asked, `HOW MANY FILES?`. This lets you specify the maximum number of files that will be "open" or in use at once. (See OPEN.) Type in an appropriate number and press (ENTER), or simply press (ENTER) and BASIC will provide for three files.

For example, if your program requires one input file and one output file, you should ask for two files.

Note: Normally, BASIC will give all your data files a record length of 256. (See **File Access Techniques**.) If you wish to set the record length of each file *individually*, use the suffix V for "Variable" after the number of files.
For example,

```
HOW MANY FILES? 3V (ENTER)
```

tells BASIC to give you three **file-buffers**, and to let you set the record length of each file when that file is first opened.

Note: Disk BASIC automatically creates a buffer for loading, saving, and merging BASIC programs. This buffer exists in RAM below any data file buffers you may request. It is always available for program I/O, regardless of how you answer the FILES? question.

After you answer the FILES question, BASIC will ask: `MEMORY SIZE?` Simply press (ENTER) without typing a number. You will then have the maximum amount of RAM available for use by BASIC.

If you will want to load and use machine-language programs or routines, you will have to protect your BASIC memory from these machine-language programs.

In such a case, respond with the highest memory address (in decimal form) you want BASIC to use for storing and executing your BASIC programs. Addresses above the number you specify will then be protected from use by BASIC.

Example:

```
MEMORY SIZE? 32000 (ENTER)
```

causes BASIC to protect addresses above 32000. If you have 16K of RAM, this means that you'll have 32767-32000 = 767 bytes protected for storing your machine-language routines.

After you answer the `MEMORY SIZE?` question, Disk BASIC will display the following information:

1. Which version of Disk BASIC you are using
2. Copyright information
3. The number of free bytes available
4. The number of concurrent files you have requested.

To exit from Disk BASIC and return to the TRSDOS READY mode, type:

```
CMD"S" (ENTER)
```

This results in a normal return to TRSDOS, without re-initialization of the system. You may recover your program if you haven't changed user memory while in TRSDOS. Use BASIC *.

Enhancements to Model III BASIC

Disk BASIC adds many features which are not disk-related. They are listed below along with abbreviated descriptions. Detailed descriptions follow in alphabetical order.

&H	Hexadecimal-constant prefix
&O	Octal-constant prefix
Abbreviations	Many commands have abbreviations
CMD"A"	Return to TRSDOS with error message
CMD"B"	Enable/Disable (BREAK)
CMD"C"	Delete spaces and remarks from a program (compression)
CMD"D"	Display directory for specified drive
CMD"E"	Display previous TRSDOS error
CMD"I"	Return a command to TRSDOS
CMD"J"	Convert calendar date
CMD"L"	Load Z-80 subroutine or program file into RAM
CMD"O"	Alphabetizes (sorts) a string array only
CMD"P"	Check printer status
CMD"R"	Start real-time clock display
CMD"S"	Normal return to TRSDOS (jump to EXIT routine)
CMD"T"	Turn off real-time clock display
CMD"X"	Cross-reference of reserved words, string variables, or strings in a program
CMD"Z"	Duplicate output to Display and Printer
DEF FN	Define BASIC-statement function
DEF USR	Define the entry point for an external machine-language routine
INSTR	Instring function; find the substring in the target string
LINE INPUT	Input a line from keyboard
MID$=	Replace portion of the target string (used on left of equals sign)
NAME	Renumber a program in RAM
USR*n*	Call external routine (n=0,1,2, . . . ,9)

&H and &O (hex and octal constants)

Often it is convenient to use hexadecimal (base 16) or octal (base 8) constants rather than their decimal counterparts. For example, memory addresses and byte values are easier to manipulate in hex form. &H and &O let you introduce such constants into your program.

&H and &O are used as prefixes for the numerals that immediately follow them:

&H*dddd*

**   *dddd* is a 1 to 4 digit sequence composed of hexadecimal numerals 0,1,...9,A,B,...,F.**

&O*ddddd*

**   *ddddd* is a sequence of octal numerals 0,1,...,7 and &O*ddddd*< = 177777 decimal.**

Note: The O can be omitted from the prefix &O. Therefore &O*ddddd*= &*ddddd*.

The constants always represent signed integers. Therefore any hex number greater than &H7FFF, or any octal number greater than &O77777, will be interpreted as a negative quantity. The following table illustrates this:

Octal	Hex	Decimal
&1	&H1	1
&2	&H2	2
&77777	&H7FFF	32767
&100000	&H8000	−32768
&100001	&H8001	−32767
&100002	&H8002	−32766
&177776	&HFFFE	−2
&177777	&HFFFF	−1

Hex and octal constants cannot be typed in as responses to an INPUT prompt or be contained in a DATA statement. Often the hex or octal constant must be enclosed in parentheses to prevent a syntax error from occurring.

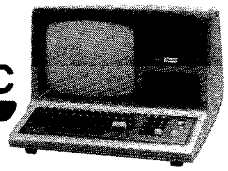
Examples

```
PRINT &H5200, &O51000
```

prints the decimal equivalent of the two constants (both equal 20992).

```
POKE &H3C00, 42
```

puts decimal 42 (ASCII code for an asterisk) into video memory address hex 3C00.



Model III Disk BASIC Abbreviations

Abbreviation	Meaning
↑	List Previous Program Line
↓	List Next Program Line
.	List Current Program Line
,	Edit Current Program Line
SHIFT ↑	List First Program Line
SHIFT ↓ Z	List Last Program Line
L*xx*	List Program Line *xx*
E*xx*	Edit Program Line *xx*
D*xx*	Delete Program Line *xx*
A*xxx*,*xxxx*	Automatic Line Numbering Beginning at Line *xxx*, Incrementing by *xxxx*.

CMD "A" Return to TRSDOS

```
CMD"A"
```

This command allows you to return to TRSDOS with an error message:

```
OPERATION ABORTED
```

Sample Use

After an input/output error occurs in a BASIC program, you might want to exit to TRSDOS and print a message.

```
CMD"A"
```

the following will be displayed:

```
OPERATION ABORTED
TRSDOS READY
```

♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦

CMD "B"
Enable/Disable BREAK Key

CMD"B", *"switch"*

switch **is either ON or OFF.** *switch* **must be enclosed in quotation marks.**

This command enables or disables the BREAK key. While the function is "OFF," the BREAK key will be ignored except during cassette or printer output or during serial input/output.

The BREAK key will remain disabled even after the program has ended. To enable the BREAK key, use the CMD"B","ON" command. Returning to TRSDOS via the CMD"S" or CMD"I" commands will also enable the BREAK key.

Examples

```
CMD"B","OFF"
```

Disables the BREAK key.

```
CMD"B","ON"
```

Returns the BREAK key to its normal function.

CMD "C"
Compress Program

CMD "C", *options*

options **may be either R (delete remarks) or S (delete spaces). If both options are omitted, remarks and spaces are deleted. If only one is omitted, only the specified action is taken.**

This command allows you to compress a program so that it requires less RAM and less storage space on diskette. You can elect to remove all remark statements (beginning with REM or ') or to delete all spaces between BASIC keywords. Spaces inside quotes will *not* be deleted.

Example

Your program reads as follows:

```
85Ø RESTORE: ON ERROR GOTO 8ØØ  'DOG PROGRAM
86Ø READ COMPANY$               'PET STORE
87Ø PRINT RIGHT$(COMPANY$,2),: GOTO 86Ø
88Ø END
```

If you want to delete the Remarks (lines 850 and 860), type in the command:

```
CMD"C",R
```

and the program will now read:

```
85Ø RESTORE: ON ERROR GOTO 8ØØ
86Ø READ COMPANY$
87Ø PRINT RIGHT$(COMPANY$,2),:GOTO 86Ø
88Ø END
```

If you then wanted to delete the spaces, type in:

```
CMD"C",S
```

and the program would read:

```
85Ø RESTORE:ONERRORGOTO8ØØ
86Ø READCOMPANY$
87Ø PRINTRIGHT$(COMPANY$,2),:GOTO86Ø
88Ø END
```

You could obtain the same results by typing:

```
CMD"C"
```

Note: Always provide the closing quotes on string literals in your BASIC program. Otherwise CMD"C" may not function properly. For example, in

```
1Ø PRINT "THIS IS A TEST"
```

the second quote should be used even though omitting it will not cause an error.

CMD"D"
Display the Directory of a Specified Drive

CMD"D:*d*"

_d_ is the drive specification

By entering the command CMD"D:*d*", you can obtain a specified diskette's directory from BASIC without returning to TRSDOS. Only unprotected, visible files will be displayed. The drive specification is not optional and must be specified for all drives, including Drive 0.

Example

If you type in the command:

```
CMD"D:1"
```

the directory for Drive 1 will be displayed.

CMD"E"
Display Previous TRSDOS error

CMD"E"

This command displays the last TRSDOS error message. If no errors have occurred prior to the command, the message `NO ERROR FOUND` will be displayed.

Example

If you have a two-drive system (0 and 1) and you type:

```
SAVE "PROGRAM:3"
```

Disk BASIC will return a `DISK I/O ERROR`. To find out what kind of I/O error occurred, type: `CMD"E"` (ENTER) and Disk BASIC will return with `DISK DRIVE NOT IN SYSTEM`.

CMD"I"
Execute TRSDOS Commands from Disk BASIC

CMD"I", *command*

command **is a string expression containing a TRSDOS command or a Z-80 program file name. If it is a string constant, it must be enclosed in quotes.**

You may execute TRSDOS commands directly from BASIC by using CMD"I".

This is similar to CMD"S", except that it lets you include a command or Z-80 program for TRSDOS to execute.

As long as BASIC is not overwritten by the execution of the program or command, control will return to BASIC; otherwise, control will return to TRSDOS. (TRSDOS commands all overlay BASIC; your Z-80 program may not if it loads *above* BASIC.)

Example

```
CMD"I","PROGRAM"
```

returns you to TRSDOS and executes the program file PROGRAM.

```
CMD"I",A$
```

returns you to TRSDOS and executes the command contained in A$.

CMD"J"
Calendar Date Conversion

CMD"J", *source, destination*

source **is a string expression containing the date to be converted. Its contents may be in either of two formats:**

A) *mm/dd/yy*

B) *-yy/ddd*

Format A gives the date in month-day-year sequence. Format B gives the day of the year (from 1 to 365 or 366 for leap years). In format B, the hyphen is required.

destination **is a string variable to contain the *converted* date. If *source* is in format A, *destination* will contain the day of year. If *source* is in format B, *destination* will contain the date in format A.**

This command converts dates back and forth between two formats: the standard month, day, year, sequence; and a year, day of year, sequence. The content of the source string determines which way the conversion goes.

Example

```
CMD"J", "11/30/80", D$
```

Returns the day of the year in D$.

```
CMD"J", "-79/300", D$
```

Returns the month, day, year, equivalent in D$ (the date for the 300th day of 1979).

Sample Program

```
10 CLEAR 50
20 LINE INPUT"ENTER FIRST DATE (MM/DD/YY) "; FD$
30 LINE INPUT"ENTER SECOND DATE (MM/DD/YY) ";SD$
40 CMD"J", FD$, D1$
50 CMD"J", SD$, D2$
60 Y1 = VAL(RIGHT$(FD$,2))
70 Y2 = VAL(RIGHT$(SD$,2))
80 J1 = VAL(RIGHT$(D1$,3))
90 J2 = VAL(RIGHT$(D2$,3))
100 S1 = Y1*365 + J1
110 S2 = Y2*365 + J2
120 PRINT "THE INTERVAL BETWEEN DATES IS";
130 PRINT ABS(S1-S2); "DAYS ";
140 PRINT "(IGNORING LEAP-YEARS)."
150 INPUT "<ENTER> TO CONTINUE"; A$
160 GOTO 20
```

CMD"L"
Load Z-80 Routine into RAM

CMD"L",*routine*

routine **is a string expression containing a file specification for a Z-80 routine or program created by the DUMP command. If *routine* is a string constant, it must be enclosed in quotes.**

CMD"L" loads a Z-80 (machine-language) routine into RAM. It would normally be used to load a Z-80 subroutine which is to be accessed directly from BASIC.

The Z-80 routine should load into high-RAM and must not overlay the memory protect area reserved when you first entered BASIC (i.e., the `MEMORY SIZE?` prompt). If you do not overlay BASIC or TRSDOS, control will return to BASIC after the program is loaded.

Example

The command:

```
CMD"L","PROG"
```

will load a program file named PROG into RAM.

```
CMD"L",P$
```

will load a program which has been specified as P$.

CMD"O"
Sort ("Order") an Array

CMD"O",*x,array (start)*

x **is an integer variable containing the number of items to be sorted.**

array (start) **specifies an array element. The** ***array*** **contains the data to be sorted, and** ***start*** **is the subscript of the first element to be sorted. The** ***array*** **must be one-dimensional, string type. The string elements in** ***array*** **may be of any length.**

This command sorts (orders) a one-dimensional string array, i.e., a list. You may sort all or part of the array, depending on the values you give to *x* and *start*.

Sample Program

```
10 CLEAR 10 * 25 + 50   'ROOM FOR 10 WORDS + EXTRA
20 DIM A$(9)            'LIST OF TEN (0-9)
30 FOR WD = 0 TO 9
40 PRINT "ENTER WORD #"; WD+1
50 INPUT A$(WD)
60 NEXT WD
70 N%=10: CMD"O", N%, A$(0)
80 PRINT "HERE IS THE SORTED LIST"
90 FOR WD=0 TO 9
```

```
100 PRINT A$(WD)
110 NEXT WD
```

CMD"P"
Check Printer Status

> CMD"P",*status*
>   *status* is a string variable

CMD"P" makes it possible for Disk BASIC to check the status of the printer.

Unlike the video display, the printer is not always available. It may be disconnected, offline, out of paper, etc. In such cases, when you try to output information to the printer, the Computer will wait until the printer becomes available. It will appear to "hang up." To regain keyboard control (and cancel the printer operation), press **BREAK**.

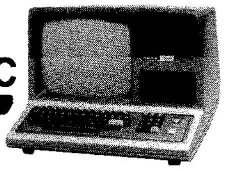
Suppose you have a program which uses printer output. If a printer is not available, you don't want the Computer to stop and wait for it to become available. Instead, you may want to print a message such as `PRINTER UNAVAILABLE` and go on to some other operation.

To accomplish this, you need to check the printer status. CMD"P" can be used to check the printer's status at any time. It returns the contents as an ASCII-coded decimal number. The specific value of this number depends upon the type of printer you are using as well as its status at any particular time. The value may then be printed or examined by the program.

Only the four most significant bits are used in this "status byte." In binary, these must be: "0011" or else the print operation will not be attempted. To check for this "go" condition, AND the status byte with 240 and compare the result with 48. The meaning of each status bit depends on which printer you use. See the printer owner's manual for bit designations.

Sample Program

```
10 CMD"P",X$
20 ST% = VAL(X$) AND 240
30 IF ST% <> 48 THEN PRINT "PRINTER UNAVAILABLE": STOP
40 PRINT "PRINTER AVAILABLE"
50 REM PROGRAM MAY NOW CONTINUE
```



CMD"R"
Turn On Clock-Display

```
CMD"R"
```

This command controls the real-time clock display in the upper-right corner of the Video Display. When it is on, the 24-hour time will be displayed and updated once each second, regardless of what program is executing.

Note: The real-time clock is always running (except during cassette or disk I/O), regardless of whether the display is on or off.

Example

To turn on the clock display type: `CMD"R"` To turn the display off, type: `CMD"T"`

CMD"S"
Return to TRSDOS

```
CMD"S"
```

To exit from Disk BASIC, returning control to TRSDOS, simply type in the command:

```
CMD"S"
```

To return to BASIC and recover your program, use `BASIC *`. However, recovery will not always be possible. See **BASIC** *.

Example

The BASIC prompt lets you know you are in Disk BASIC.

```
READY
>
```

To exit, type in:

```
CMD"S"
```

and the TRSDOS prompt will appear.

```
TRSDOS READY
```

♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦ ♦

CMD"T" Turn Off Clock-Display

CMD"T"

This command turns off the real-time clock display function.

However, the clock continues to run.

Example

To stop the clock display update type: `CMD"T"`

To start the display, type: `CMD"R"`

CMD "X" Cross-reference of Program Lines

CMD "X", *target*

***target* is either a BASIC reserved word (such as PRINT) or a string-literal. If it is a reserved word, it must not be enclosed in quotes; if it is a string-literal, it must be enclosed in quotes.**

This command finds all occurrences of a reserved word or other string literal in the resident program. The ''finds'' are listed on the display as five-digit line numbers.

To search for any BASIC reserved word (including reserved arithmetic operators), use the keyword as-is. To search for anything else (including variable-names and text), enclose the text inside quotes.

For example, suppose you have the following program in memory:

```
10 PRINT "THIS IS A TEST"
20 INPUT "PRESS <ENTER> FOR THE NEXT PRINT MESSAGE"; Z$
30 A = A + 1
40 PRINT "+++++++"
```

`CMD "X", PRINT` will find all occurrences of PRINT, except for cases where PRINT was part of a quoted string: lines 10 and 40.

`CMD "X", "PRINT"` will find all occurrences of ''PRINT'' as a string literal: line 20.

`CMD "X", +` will list line 30, but `CMD "X", "+"` will list line 40. `CMD "X", "A"` will list lines 10, 20, and 30. Notice that variables and text are both treated as string literals.

CMD "Z"
Duplicate Output to Video and Printer

CMD"Z", *"switch"*

***switch* is either ON or OFF. *switch* must be enclosed in quotation marks.**

This command enables or disables dual video/printer output. While the function is ''ON,'' all video output is copied to the printer, and all printer output is copied to the video. (The printer *must* be on-line when you turn dual output ''ON.'')

Video and printer output may differ due to intrinsic differences in the printer and video devices.

Examples

```
CMD "Z", "ON"
```

Turns dual video/printer output on.

```
CMD"Z", "OFF"
```

Turns dual video/printer output off.

DEF FN
Define Function

DEF FN *function name (argument-1, . . .) = formula*

function name **is any valid variable name.**

argument-1 **and subsequent arguments are used in defining what the function does.**

formula **is an expression usually involving the argument(s) passed on the left side of the equals sign.**

The DEF FN statement lets you create your own function. That is, you only have to call the new function by name, and the associated operations will automatically be performed. Once a function has been defined with the DEF FN statement, you can call it simply by inserting FN in front of *function name*. You can use it exactly as you might use one of the built-in functions, like SIN, ABS, and STRING$.

The type of variable used for *function name* determines the type of value the function will return. For example, if *function name* is single precision, then that function will return a single-precision value, regardless of the precision of the arguments.

The particular variables you use as arguments in the DEF FN statement *(argument-1, . . .)* are not assigned to the function. When you call the function later, any variable name of the same type can be used.

Furthermore, using a variable as an argument in a DEF FN statement has no effect on the value of that variable. So you can use that particular variable in another part of your program without worrying about interference from DEF FN.

The function can be defined with no arguments at all, if none are required. For example:

```
DEF FNR = RND (90) + 9
```

defines a function to return a random value between 10 and 99.

Examples

```
DEF FNR(A,B) = A + INT((B - (A - 1)) * RND(0))
```

This statement defines function FNR which returns a random number between integers A and B. The values for A and B are passed when the function is "called," i.e., used in a statement like:

```
Y = FNR(R1, R2)
```

If R1 and R2 have been assigned the values 2 and 8, this line would assign a random number between 2 and 8 to Y.

```
DEF FNL$(X) = STRING$(X, "-")
```

Defines function FNL$ which returns a string of hyphens, X characters long. The value for X is passed when the function is called:

```
PRINT FNL$(3)
```

This line prints a string of 30 hyphens.

Here's an example showing DEF FN used for a complex computation — in double-precision.

```
DEF FNX#(A#, B#) = (A# - B#) * (A# - B#)
```

Defines function FNX# which returns the double-precision value of the square of the difference between A# and B#. The values for A# and B# are passed when the function is called:

```
S# = FNX#(A#, B#)
```

We assume that values for A# and B# were assigned elsewhere in the program.

Sample Program

```
710 DEF FNV(T) = (1087 + SQR(273 + T))/16.52
720 INPUT "AIR TEMPERATURE IN DEGREES CELSIUS"; T
730 PRINT "THE SPEED OF SOUND IN AIR OF" T "DEGREES
    CELSIUS IS" FNV(T) "FEET PER SECOND."
```

DEFUSR
Define Point of Entry for USR Routine

DEFUSR*n* = *address*

n equals one of the digits 0, 1,...,9; if *n* is omitted, 0 is assumed. *address* specifies the entry address to a machine-language routine. *address* must be in the range [−32768,32767]. *address* may be any numeric expression or constant from −32768 to 32767.

DEFUSR lets you define the entry points for up to 10 machine-language routines. In non-Disk BASIC, the addresses were POKEd into RAM. This POKE method cannot be used in Disk BASIC.

Examples

```
DEFUSR3 = &H7D00
```

assigns the entry point X'7D00', 32000 decimal, to the USR3 call. When your program calls USR3, control will branch to your subroutine beginning at X'7D00'.

```
DEFUSR = (BASE + 16)
```

assigns start address (BASE + 16) to the USR0 routine.

Note: When decimal addresses are given, they are evaluated as signed two-byte integers. So, for addresses above 32767, use *desired decimal address −65536*. See **USR*n***.

INSTR
Search for Specified String

INSTR *(position, string 1, string 2)*

position specifies the position in *string 1* where the search is to begin. *position* is optional; if it is not supplied, search automatically begins at the first character in *string 1*. (Position 1 is the first character in *string 1*.)

***string 1* is the string to be searched.**

***string 2* is the substring you want to search for.**

This function lets you search through a string to see if it contains another string. If it does, INSTR returns the starting position of the substring in the target string; otherwise, zero is returned. Note that the entire substring must be contained in the search string, or zero is returned. Also, note that INSTR only finds the first occurrence of a substring at the position you specify.

Examples

In these examples, A$ = "LINCOLN":

```
INSTR(A$, "INC")
```

returns a value of 2.

```
INSTR (A$, "12")
```

returns a zero.

```
INSTR(A$, "LINCOLNABRAHAM")
```

returns a zero. For a slightly different use of INSTR, look at

```
INSTR (3, "1232123", "12")
```

which returns 5.

Sample Program

This program gets search and target text from the keyboard, then locates all occurrences of the target text in the search text. Line 90 is just for "show."

```
10 CLEAR 1000
20 CLS
30 INPUT "SEARCH TEXT"; S$
40 INPUT "TARGET TEXT"; T$
45 CLS
50 C = 0 : P = 1 'P = POSITION, C = COUNT
60 F = INSTR(P,S$,T$)
70 IF F = 0 THEN 120
80 C = C + 1
90 PRINT @0,LEFT$(S$,F-1) + STRING$(LEN(T$),191) +
   RIGHT$(S$,LEN(S$)-F-LEN(T$)+1)
100 P = F + LEN(T$)
110 IF P <= LEN(S$) - LEN(T$) + 1 THEN 60
120 PRINT "FOUND "; C; "OCCURRENCES"
```

LINE INPUT
Input a Line from Keyboard

LINE INPUT *"prompt" ;variable*

***prompt* is a prompting message.**

***variable* is the name that will be assigned to the line you type in.**

LINE INPUT (or LINEINPUT — the space is optional) is similar to INPUT, except:

- The Computer will not display a question mark when waiting for your operator's input.
- Each LINE INPUT statement can assign a value to just one variable.
- Commas and quotes your operator can use as part of the string input.
- Leading blanks are not ignored — they become part of *variable*.
- The only way to terminate the string input is to press (ENTER).

LINE INPUT is a convenient way to input string data without having to worry about accidental entry of delimiters (commas, quotation marks, colons, etc.). The (ENTER) key serves as the only delimiter. If you want anyone to be able to input information into your program without special instructions, use the LINE INPUT statement.

Some situations require that you input commas, quotes and leading blanks as part of the data. LINE INPUT serves well in such cases.

Examples

```
LINE INPUT A$
```

Input A$ without displaying any prompt.

```
LINE INPUT "LAST NAME, FIRST NAME? ";N$
```

Displays a prompt message and inputs data. Commas will not terminate the input string, as they would in an input statement.

Sample Program

```
200 REM  CUSTOMER SURVEY
205 CLEAR 1000
207 PRINT
```

```
210 LINE INPUT "TYPE IN YOUR NAME "; A$
220 LINE INPUT "DO YOU LIKE YOUR COMPUTER? "; B$
230 LINE INPUT "WHY? "; C$
235 PRINT
240 PRINT A$ : PRINT
250 IF B$= "NO" THEN 270
260 PRINT "I LIKE MY COMPUTER BECAUSE "; C$ :END
270 PRINT "I DO NOT LIKE MY COMPUTER BECAUSE "; C$
```

Notice that when line 210 is executed, a question mark is not displayed after the statement, ''Type in your name.'' Also, notice on line 230 you can answer the question ''Why'' with a statement full of delimiters, commas and quotes.

MID$ =
Replace Portion of String

MID$ *(oldstring, position, length) = replacement-string*

oldstring **is the variable-name of the string you want to change.**

position **is the numeric expression specifying the position of the first character to be changed.**

length **is a numeric expression specifying the number of characters to be replaced.**

replacement-string **is a string expression to replace the specified portion of oldstring.**

Note: If *replacement-string* is shorter than *length*, then the entire *replacement-string* will be used.

This statement lets you replace any part of a string with a specified new string, giving you a powerful string editing capability.

Note that the length of the resultant string is always the same as the original string.

Examples

A$ = ''LINCOLN'' in the examples below:

```
MID$(A$, 3, 4) = "12345": PRINT A$
```

which returns LI1234N.

```
MID$(A$, 1, 2) = "": PRINT A$
```

which returns LINCOLN.

```
MID$(A$, 5) = "12345": PRINT A$
```

returns LINC123.

```
MID$(A$, 5) = "01": PRINT A$
```

returns LINC01N.

```
MID$(A$, 1, 3) = "***": PRINT A$
```

returns ***COLN.

Sample Program

```
770 CLS: PRINT: PRINT
780 LINE INPUT "TYPE IN A MONTH AND DAY MM/DD, "; S$
790 P = INSTR(S$, "/")
800 IF P = 0 THEN 780
810 MID$(S$, P, 1) = CHR$(45)
820 PRINT S$ " IS EASIER TO READ, ISN'T IT?"
```

This program uses INSTR to search for the slash (''/''). When it finds it (if it finds it), it uses MID$= to substitute a '' – '' (CHR$(45)) for it.

NAME
Renumber the Current Program

NAME ***newline, startline, increment***

newline **specifies the new line number of the first line to be renumbered. If omitted, 10 is used.**

startline **specifies the line number in the original program where renumbering will start. If omitted, the entire program will be renumbered.**

increment **specifies the increment to be used between each successive line number. If omitted, 10 is used.**

Examples

```
NAME
```

Renumbers the entire program: 10, 20, 30, . . .

```
NAME 6000,5000,100
```

Renumbers all lines numbered from 5000 up; the first renumbered line will become 6000, and the following lines will be incremented by 100. All line references within your program will be renumbered also.

USRn
Call to User's External Subroutine

USR*n* *(nmexp)*

where *n* specifies one of ten available USR calls, *n*=0,1,2,...,9. If *n* is omitted, zero is assumed.

***nmexp* is an integer from −32768 to 32767 and is passed as an integer argument to the routine.**

These functions (USR0 through USR9) transfer control to machine-language routines previously defined with DEFUSR*n* statements.

When a URS call is encountered in a statement, control goes to the address specified in the DEFUSR*n* statement. This address specifies the entry point to your machine-language routine.

Note: If you call a USR*n* routine before defining the routine entry point with DEFUSR*n*, an ILLEGAL FUNCTION CALL error will occur.

You can pass one argument and retrieve one output value directly via the USR argument; or you can pass and retrieve arguments indirectly via POKE and PEEK statements.

Example

```
10 DEFUSR1=&H7D00
20 REM...MORE PROGRAM LINES HERE
100 A=USR1(X)
```

The effect of this sequence is to:

1. Define USR as a routine with an entry point at hex 7D00 (line 10).
2. Transfer control to the routine; the value X can be passed to the routine if the routine makes the CALL described below (line 100).
3. When the routine returns to BASIC, the variable A may contain the value passed back from the routine (if your routine makes the JUMP described below); otherwise A will be assigned the value of X (line 100).

Passing arguments to and from USR routines

There are several ways to pass arguments back and forth between your BASIC main program and your USR routines: the two major ways are listed below.

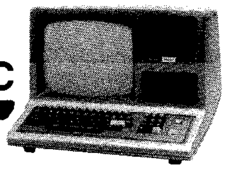
1. POKE the argument(s) into fixed RAM locations. The machine-language routine can then access these values and place results in other RAM locations. When the routine returns control to BASIC, your program can PEEK into these addresses to pick up the "output" values. **This is the only way to pass two or more arguments back and forth**.
2. Pass one argument to the routine as the argument in the USRn call, then use special ROM calls to access this argument and return a value to BASIC. **This method is limited to sending one argument and returning one value** (both are integers).

ROM Calls

CALL 0A7FH	Puts the USR argument into the HL register pair; H contains MSB, L contains LSB. This CALL should be the first instruction in your USR routine.
JP 0A9AH	Use this JUMP to return to BASIC; the integer in HL becomes the output of the USR call. If you don't care about returning HL, then execute a simple RETurn instruction instead of this JUMP.

Listed below is an assembled program to white out the display (an "inverse" CLEAR key!). Don't type it in. Type in the BASIC program that follows it.

```
                 00100 ;
                 00110 ; ZAP OUT SCREEN USR FUNCTION
                 00120 ;
7D00             00130            ORG      7D00H
                 00140 ;
                 00150 ; EQUATES
                 00160 ;
3C00             00170 VIDEO      EQU      3C00H          ;START OF VIDEO RAM
00BF             00180 WHITE      EQU      0BFH           ;ALL WHITE GRAPHICS BYTE
03FF             00190 COUNT      EQU      3FFH           ;NUMBER OF BYTES TO MOVE
                 00200 ;
                 00210 ; PROGRAM CHAIN MOVES X'BF' INTO ALL OF VIDEO RAM
                 00220 ;
```



```
7D00 21003C     00230 ZAP      LD      HL,VIDEO       ;SOURCE ADDRESS
7D03 36BF       00240          LD      (HL),WHITE     ;PUT OUT 1ST BYTE
7D05 11013C     00250          LD      DE,VIDEO+1     ;DESTINATION ADDRESS
7D08 01FF03     00260          LD      BC,COUNT       ;NUMBER OF ITERATIONS
7D0B EDB0       00270          LDIR                   ;DO IT TO IT!!!
                00280 ;
7D0D C9         00290          RET                    ;RETURN TO BASIC
7D00            00300          END     ZAP
```

This routine can be POKEd into RAM and accessed as a USR routine. First start BASIC and answer the MEMORY SIZE question with 31999. Then run the program.

```
100 '          PROGRAM: USR1
110 ' EXAMPLE OF A USER MACHINE LANGUAGE FUNCTION
115 ' DEPRESS THE '@' KEY WHILE NUMBERS ARE PRINTING TO STOP
120 '
130 ' ******* POKE MACHINE PROGRAM INTO MEMORY *******
140 '
150 DEFUSR1 = &H7D00
160 FOR X = 32000 TO 32013   '7D00 HEX EQUAL 32000 DECIMAL
170      READ A
180      POKE X, A
190 NEXT X
192 '
194 ' ******* CLEAR SCREEN & PRINT NUMBERS 1 THRU 100 *******
196 '
200 CLS
205 PRINT TAB(15); "WHITE-OUT USER ROUTINE": PRINT
210 FOR X = 1 TO 100
220      PRINT X;
225      A$ = INKEY$: IF A$ = "@" THEN END
230 NEXT X
240 '
250 ' ******* JUMP TO WHITE-OUT SUBROUTINE *******
260 '
270 X = USR1 (0)
280 FOR X = 1 TO 1000: NEXT X   'DELAY LOOP
290 GOTO 200
300 '
310 ' ******* DATA IS DECIMAL CODE FOR HEX PROGRAM *******
320 '
330 DATA 33,0,60,54,191,17,1,60,1,255,3,237,176,201
```

Run the program. An equivalent BASIC white out routine takes a long time by comparison!

Disk-Related Features

Disk BASIC provides a powerful set of commands, statements and functions relating to disk I/O under TRSDOS. These fall into two categories:

1. File manipulation: dealing with files as units, rather than with the distinct records the files contain.
2. File access: preparing data files for I/O; reading and writing to the files.

Under the heading, **File Manipulation**, we will discuss the following commands.

KILL	Delete a program or data file from the disk
LOAD	Load a BASIC program from disk
MERGE	Merge an ASCII-format BASIC program on disk with one currently in RAM
RUN"*program*"	Load and execute a BASIC program stored on disk
SAVE	Save the resident BASIC program on disk

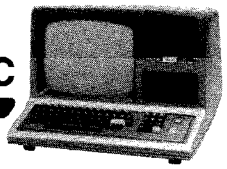
Under the heading, **File Access**, we will discuss the following statements and functions.

Statements

OPEN	Open a file for access (create the file if necessary)
CLOSE	Close access to the file
INPUT #	Read from disk, sequential mode
LINE INPUT#	Read a line of data, sequential mode
PRINT#	Write to disk, sequential mode
FIELD	Assign field sizes and names to random-access file buffer
GET	Read from disk, random access mode
PUT	Write to disk, random access mode
LSET	Place value in specified buffer field, add blanks on the right to fill field
RSET	Place value in specified buffer field, add blanks on the left to fill field

Functions

CVD	Restore double-precision number to numeric form after GETting from disk
CVI	Restore integer to numeric form after GETting from disk
CVS	Restore single-precision number to numeric form after GETting from disk
EOF	Check to see if end of file encountered during read
LOC	GET current record number.



LOF	Return number of last record in file
MKD$	Convert double-precision number to string so it can be PUT on disk
MKI$	Convert integer to string so it can be PUT on disk
MKS$	Convert single-precision number to string so it can be PUT on disk

File Manipulation

KILL
Delete a File from the Disk

KILL *exp$*

> ***exp$* defines a file specification for an existing file.**

This command works like the TRSDOS KILL command — see **TRSDOS Library Commands.**

Example

```
KILL"OLDFILE/BAS.PSW1"
```

deletes the file specified from the first drive which contains it.

Do not KILL an open file, or you may destroy the contents of the diskette. (First, CLOSE the open file.)

LOAD
Load BASIC Program File from Disk

LOAD *exp$* [,R]

> **where *exp$* defines a filespec for a BASIC program file stored on disk.**
>
> **R tells BASIC to RUN the program after it is loaded.**

This command loads a BASIC program file into RAM; if the R option is used, BASIC will proceed to RUN the program automatically; otherwise, BASIC will return to the command mode.

LOAD without the R option clears all variables and closes all open files. LOAD with the R option clears all variables but does not close the open files.

LOAD with the R option is equivalent to the command RUN *exp$*,R. Either of these commands can be used inside programs to allow program chaining—one program calling another, etc.

Example

```
LOAD"PROG1/BAS:2"
```

Clears resident BASIC program and loads PROG1/BAS from Drive 2; returns to BASIC command mode.

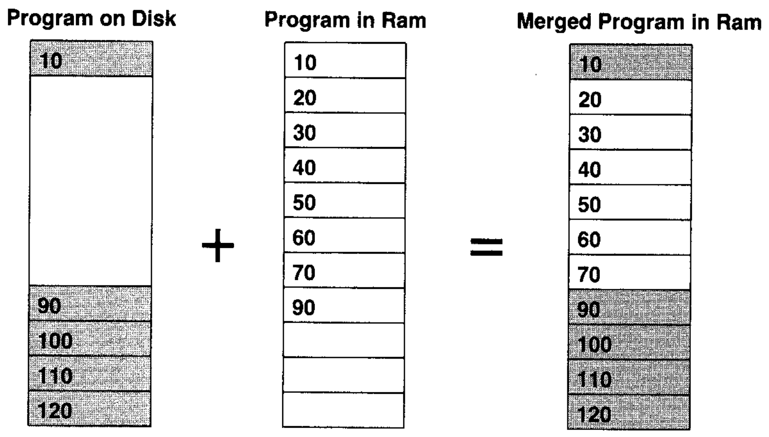
MERGE
Merge Disk Program with Resident Program

MERGE *exp$*

**	*exp$* is file specification for an ASCII-format BASIC disk file, e.g., a program saved with the A-option.**

MERGE is similar to LOAD—except that the resident program is not erased before the new program *exp$* is loaded. Instead, the new program is merged into the resident program.

That is, program lines in *exp$* will simply be inserted into the resident program in sequential order. If line numbers in *exp$* coincide with line numbers in the resident program, the resident lines will be replaced by those from *exp$*.



Sample Use

Save this program in ASCII format.

```
1000 REM . . . SUBROUTINE TO SAY HELLO
1010 PRINT "HELLO!"
1020 RETURN
```

Type NEW (ENTER), then type in this program.

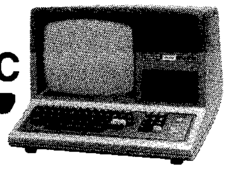
```
100 CLS
110 PRINT "LET'S CALL THE SUBROUTINE . . ."
120 PRINT "DIALING NOW . . ."
130 FOR I=1 TO 1000 : NEXT
140 GOSUB 1000
150 PRINT "BACK FROM SUBROUTINE."
160 END
```

Now type MERGE *"file"* using the file name given to the first file. List the program. Then run it.

RUN"program"
Load and Execute a Program from Disk

RUN *file*[,R]

file **is the name of a BASIC program file. It is a string expression. (If a string constant is used, it must be enclosed in quotes.) The ,R option causes BASIC to leave open files open. If omitted, open files are closed before the program is run.**



This command loads and executes a BASIC program stored on disk. It may be used inside a program to allow chaining (one program calling another).

Examples

```
RUN "PROG"
```

Loads and executes PROG (all open files are closed first).

```
A$="NEWPROG"
RUN A$, R
```

Loads and executes NEWPROG (all open files remain open).

SAVE
Save Program onto Disk

SAVE *file* [,A]

> ***file*** **is the name of a BASIC program file. It is a string expression. (If a string constant is used, it must be enclosed in quotes.)**
>
> **A causes the file to be stored in ASCII rather than compressed format.**

This command lets you save your BASIC programs on disk. You can save the program in compressed or ASCII format.

Using compressed format takes up less disk space and is faster during both SAVES and LOADS. Using the ASCII option makes it possible to do certain things that cannot be done with compressed format BASIC files.

For example:

- The MERGE command requires that the disk file be in ASCII form.
- Programs which read in other programs as data will typically require that the data programs be stored in ASCII.
- The TRSDOS command APPEND also requires that disk files be in ASCII form.

Examples

```
SAVE"FILE1/BAS. JOHNQDOE:3"
```

saves the resident BASIC program in compressed format with the file name FILE1, extension /BAS, password .JOHNQDOE; the file is placed on Drive :3.

```
SAVE"MATHPAK/TXT",A
```

saves the resident program in ASCII form, using the name MATHPAK/TXT, on the first nonwrite-protected diskette.

Upon completion of a SAVE, BASIC returns in the command mode.

File Access

This section is divided into four parts:

1. Creating files and assigning buffers — OPEN and CLOSE
2. Statements and functions
3. Sequential I/O techniques
4. Random I/O techniques

If this is your first experience with disk file access, you should concentrate on parts 1, 3 and 4, perhaps just skimming through part 2 to get a general idea of how the functions and statements work. Later you can go back to part 2 and learn the details of statement and function syntax.

Creating Files and Assigning Buffers

During the initialization dialog, you type in a number in response to HOW MANY FILES? The number you type in tells BASIC how many buffers to create to handle your disk accesses (reads and writes).

Each buffer is given a number from 1 to 15. If you type:

```
HOW MANY FILES? 3V (ENTER)
```

BASIC sets aside 3 buffers, numbered 1,2,3.

You can think of a buffer as a waiting area that data must pass through on the way to and from the disk file. When you want to access a particular file, you must tell BASIC which buffer to use in accessing that file. You must also tell BASIC what kind of access you want — sequential output, sequential input, or random input/output.

All this is done with the OPEN statement, and "undone" with the CLOSE statement.

OPEN
Open a File

OPEN ***mode, buffer, file, record-length***

mode **is a string expression containing one of the following:**

I Sequential input starting at the first record. If the file is not found, it will be created.

- O Sequential output starting at the first record. If the file is not found, it will be created.
- E (Extend) Sequential output starting at end of file. If the file is not found, it will be created.
- R Random input/output. If the file is not found, it will be created.

If *mode* is a constant, it must be enclosed in quotes.

buffer is a numeric expression specifying which buffer is to be used.

file is a string expression containing the file specification. If a constant is used, it must be enclosed in quotes.

record-length is a numeric expression from 0 to 256 specifying the logical record length. 0 is the same as 256. This option may only be used if variable-length records were requested during initialization (How Many Files?). If record-length is omitted, 256 is used. *record-length* is used with Random access *only.*

This statement lets you create a file, write data into it, update it, and read it. For details on file access, see **Methods of Access** later in this section.

If *file* includes a drive specification, BASIC will use only the specified drive. If no drive is specified, BASIC will search for a matching file, starting with the master drive (usually Drive 0).

Examples

```
OPEN "O", 1, "DATAFILE"
```

Opens DATAFILE (creates it if it doesn't already exist) for sequential output. Output will be done through buffer #1. Records will be 256 bytes long. Since the "O" mode is specified, output will start at the first record in the file.
If "E" is used instead of "O", output will start at the end of the file.

```
OPEN "R", 2, "PAYROLL/A:1", 64
```

Opens/creates PAYROLL/A for random input/output. Access will be through buffer #2. Records will be 64 bytes long (if BASIC was initialized for variable-length records).

```
BUFFER = 3: FILE$ = "DATA": RECLN = 128
OPEN "R", BUFFER, FILE$, RECLN
```

Opens/creates DATA for random input/output. Access will be through buffer #3. Records will be 128 bytes long (if BASIC was initialized for variable-length records).

CLOSE
Close Access to the File

CLOSE [*nmexp* [,*nmexp*. . .]]

nmexp has a value from 1 to 15, and refers to the file's buffer number (assigned when the file was opened). If _nmexp_ is omitted, all open files will be closed.

This command terminates access to a file through the specified buffer(s). If *nmexp* has not been assigned in a previous OPEN statement, then

CLOSE *nmexp*

has no effect.

Examples

```
CLOSE 1,2,8
```

Terminates the file assignments to buffers 1, 2 and 8. These buffers can now be assigned to other files with OPEN statements.

```
CLOSE FIRST%+COUNT%
```

Terminates the file assignment to the buffer specified by the sum (FIRST% + COUNT%).

Do not remove a diskette which contains a file opened for writing (mode = O, E, or R). First close the file. This is because the last 256 bytes of data may not have been written to disk yet. Closing the file will write the data, if it hasn't already been written.

Any modification to the resident program (NEW, editing, LOAD, MERGE, etc.) will cause open files to be closed.

INPUT#
Sequential Read from Disk

INPUT# *nmexp*, *var*[,*var* . . .]

where *nmexp* specifies a sequential input file buffer, *nmexp*=1,2,...,15.

var is the variable name to contain the data from the file.

This statement inputs data from a disk file. The data is input sequentially. That is, when the file is first opened, a pointer is set to the beginning of the file. Each time data is input, the pointer advances. To start over reading from the beginning of the file, you must close the file and re-open it.

INPUT# doesn't care how the data was placed on the disk — whether a single PRINT# statement put it there, or whether it required 10 different PRINT# statements. What matters to INPUT# are the positions of the terminating characters and the EOF marker.

To INPUT# data successfully from disk, you need to know ahead of time what the format of the data is. Here is a description of how INPUT# interprets the various characters it encounters when reading data.

When inputting data into a variable, BASIC ignores leading blanks; when the first non-blank character is encountered, BASIC assumes it has encountered the beginning of the data item.

The data item ends when a terminating character is encountered or when a terminating condition occurs. The particular terminating characters vary, depending on whether BASIC is inputting to a numeric or string variable.

Special Note

Here's an important exception to keep in mind in reading the following material.

When ENTER (a carriage return) is preceded by ↓ (a line feed), the ENTER is not taken as a terminator. Instead, it becomes a part of the data item (string variable) or is simply ignored (numeric variable).

(To enter the ↓ character from the keyboard, press the down-arrow character. To enter the ENTER character, press ENTER.)

This exception applies to all cases noted below where ENTER is said to be a terminator.

Numeric Input

Suppose the data image on disk is

```
1.234 -33 27 (ENTER)
```

(ENTER) denotes a carriage-return character (ASCII code decimal 13).

Then the statement

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

or the sequence of statements

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

will assign the values as follows:

A = 1.234
B = −33
C = 27

This works because blanks and (ENTER) serve as terminators for input to numeric variables. The blank before 1.234 is a "leading blank," therefore it is ignored. The blank after 1.234 is a terminator; therefore BASIC starts inputting the second variable at the − character, inputs the number −33, and takes the next two blanks as terminators. The third input begins at the 2 and ends with the 7.

String Input

When reading data into a string variable, INPUT ignores all leading blanks; the first non-blank character is taken as the beginning of the data item.

If this first character is a double-quote ("), then INPUT will evaluate the data as a quoted string: it will read in all subsequent characters up to the next double-quote. Commas, blanks, and (ENTER) characters will be included in the string. The quotes themselves do not become a part of the string.

If the first character of the string item is not a double-quote, then INPUT will evaluate the data as an unquoted string: it will read in all subsequent characters up to the first comma, or (ENTER). If double-quotes are encountered, they will be included in the string.

For example, if the data on disk is:

PECOS, TEXAS"GOOD MELONS"

Then the statement

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

would assign values as follows:

A$ = PECOS

B$ = TEXAS ''GOOD MELONS''

C$ = null string

If a comma is inserted in the data image before the first double quote, C$ will get the value, GOOD MELONS.

These are very simple examples just to give you an idea of how INPUT works. However, there are many other ways to input data—different terminators, different target variable types, etc.

Rather than taking a shotgun approach and trying to cover them all, we'll give a generalized description of how input works and what the terminating characters and conditions are, and then provide several examples.

When BASIC encounters a terminating character, it scans ahead to see how many more terminating characters it can include with the first terminator. This ensures that BASIC will begin looking for the next data item at the correct place.

The list below defines the various terminating sets INPUT# will look for. It will always try to take-in *the largest set possible.*

Numeric-input terminator sets

end of file encountered
255th data character encountered
,(comma)
(ENTER)
(ENTER) (↓)
[. . .] [(ENTER)]
[. . .] [(ENTER) (↓)]

Quoted-string terminator sets
end of file encountered
255th data character encountered
'' (double quote)
'' [. . .] [,]
'' [. . .] [(ENTER)]
'' [. . .] [(ENTER) (↓)]

Unquoted-string terminator sets

end of file encountered
255th data character encountered
,
(ENTER) [(↓)]

Figure 13 describes how INPUT# assigns data to a variable.

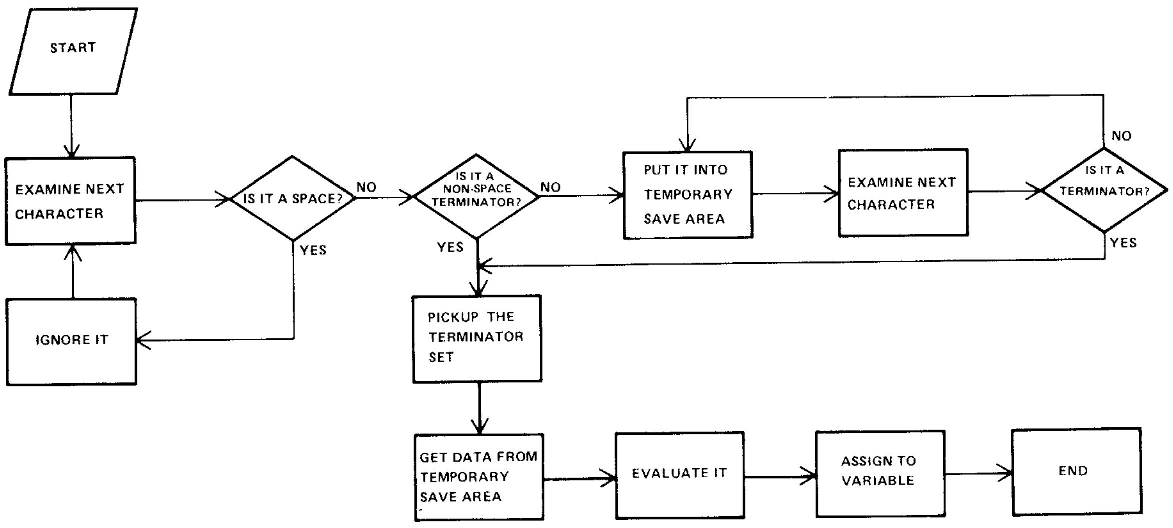


Figure 13. Input process.

The following table shows how various data images will be read-in by the statement:

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

Ex.#	Image on disk	Values assigned

1	123.45 (ENTER)(↓) 8.2E4 7000(ENTER)	A=123.45
		B=82000
		C=7000

2	3(↓)(ENTER) 4 (ENTER)5 (ENTER) A12 eof	A=34
		B=5
		C=0

3	1,,2,3,4 (ENTER)	A=1
		B=0
		C=2

4	1,3, eof	A=1
		B=3
		C=0 eof error

(eof = end of file):

In Example 2 above, why does variable C get the value 0? When the input reaches the end of file, it terminates that last data item, which then contains "A12." This is evaluated by a routine just like the BASIC VAL function—which returns a zero since the first character of "A12" is a non-numeric.

In Example 3, when INPUT# goes looking for the second data item, it immediately encounters a terminator (the comma); therefore, variable B is given the value zero.

The following table shows how various data images on disk will be read by the statement:

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

Ex.#	Image on disk	Values assigned
1	`"ROBERTS,J."ROBERTS,M,N` eof	A$:ROBERTS,J. B$:ROBERTS,M.N.
2	`ROBERTS,J.,   ROBERTS,M.N.` ENTER	A$:ROBERTS B$:J.
3	`THE WORD "QUO",12345.789` ENTER	A$:THE WORD "QUO" B$:12345.789
4	`BYTE`↓ ENTER `UNIT OF MEMORY` eof	A$:BYTE↓ENTER UNIT OF MEMORY B$:null (eof error)

In Example 3, the first data item is an unquoted string, therefore, the double-quotes are not terminators, and become part of A$.

In Example 4, the ENTER is preceded by an ↓, therefore it does not terminate the first string; both ↓ and ENTER are included in A$.

LINE INPUT#
Read a Line of Text from Disk

LINE INPUT#*nmexp,var$*

where *nmexp* specifies a sequential output file buffer, *nmexp*=1,2,...,15.

***var$* is the variable name to contain the string data.**

Similar to LINE INPUT from keyboard, this statement reads a "line" of string data into *var$*. This is useful when you want to read an ASCII-format BASIC program file as data, or when you want to read in data without following the usual restrictions regarding leading characters and terminators.

LINE INPUT (or LINEINPUT — the space is optional) reads everything from the first character up to:

1. an (ENTER) character which is not preceded by (↓)
2. the end of file
3. the 255th data character (this 255 character is included in the string)

Other characters encountered — quotes, commas, leading blanks, (↓) (ENTER) pairs — are included in the string.

For example, if the data looks like:

```
10 CLEAR 500 (ENTER)
20 OPEN"I",1,"PROG" (ENTER)
.
.
.
```

then the statement

```
LINEINPUT#1,A$
```

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

PRINT#
Sequential Write to Disk File

PRINT#*nmexp*,[USING *format$*;] *exp*[*p exp*. . .]

where *nmexp* specifies a sequential output file buffer, *nmexp*=1,2,...,15.

***format$* is a sequence of field specifiers used with the USING option.**

***p* is a delimiter placed between every two expressions to be PRINTed to disk; either a semi-colon or comma can be used (semi-colon is preferable).**

***exp* is the expression to be evaluated and written to disk.**

This statement writes data sequentially to the specified file. When you first open a file for sequential output, a pointer is set to the beginning of the file, therefore your first PRINT# places data at the beginning of the file. At the end of each PRINT# operation, the pointer advances, so the values are written in sequence.

A PRINT# statement creates a disk image similar to what a PRINT to display creates on the screen. Remember this, and you'll be able to set up your PRINT# list correctly for access by one or more INPUT statements.

PRINT# does not compress the data before writing it to disk; it writes an ASCII-coded image of the data.

For example, if A = 123.45

```
PRINT#1,A
```

will write a nine-byte character sequence onto disk:

```
 123.45  (ENTER)
```

The punctuation in the PRINT list is very important. Unquoted commas and semi-colons have the same effect as they do in regular PRINT to display statements.

For example, if A = 2300 and B = 1.303, then

```
PRINT#1,A,B
```

places the data on disk as

```
 2300                1.303  (ENTER)
```

The comma between A and B in the PRINT# list causes 10 extra spaces in the disk file. Generally you wouldn't want to use up disk space this way, so you should use semi-colons instead of commas.

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

writes the data as:

```
 2300   1.303 (ENTER)
```

PRINT# with numeric data is quite straightforward—just remember to separate the items with semi-colons.

PRINT# with string data requires more care, primarily because you have to insert delimiters so the data can be read back correctly. In particular, you must separate string items with explicit delimiters if you want to INPUT# them as distinct strings.

For example, suppose:

`A$="JOHN Q. DOE"` and `B$="100-01-001"`

Then:

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

would produce this image on disk:

```
JOHN Q. DOE100-01-001 (ENTER)
```

which could not be INPUT back into two variables.

The statement:

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

would produce:

```
JOHN Q. DOE, 100-01-001
```

which could be INPUT# back into two variables.

This method is adequate if the string data contains no delimiters — commas or ENTER — characters. But if the data does contain delimiters or leading blanks that you don't want to ignore, then you must supply explicit quotes to be written along with the data. For example, suppose `A$="DOE, JOHN Q."` and `B$="100-01-001"`

If you use

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

the disk image will be:

```
DOE, JOHN Q.,100-01-001 ENTER
```

When you try to input this with a statement like

```
INPUT#2,A$,B$
```

A$ will get the value DOE, and B$ will get JOHN Q. — because of the comma after DOE in the disk image.

To write this data so that it can be input correctly, you must use the CHR$ function to insert explicit double quotes into the disk image. Since 34 is the decimal ASCII code for double quotes, use CHR$(34) as follows:

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

this produces the disk image

```
"DOE, JOHN Q."100-01-001 ENTER
```

which can be read with a simple

```
INPUT#2,A$B$
```

Note: You can also use the CHR$ function to insert other delimiters and control codes into the file, for example:

CHR$(10)	↓ Line Feed
CHR$(13)	carriage return (ENTER character)
CHR$(11) or CHR$(12)	line-printer top-of-form

USING Option

This option makes it easy to write files in a carefully controlled format.

For example, suppose:

```
A$="LUDWIG"
```

```
B$="VAN"
C$="BEETHOVEN"
```

Then the statement

```
PRINT#1,USING"!. !. %  %";A$;B$;C$
```

would write the data in nickname form:

```
L.V.BEET <ENTER>
```

(In this case, we didn't want to add any explicit delimiters.) See the PRINT USING description in the *LEVEL II BASIC Reference Manual* for a complete explanation of the field-specifiers.

Random Access Statements

FIELD
Organize a Random File-Buffer into Fields

FIELD *nmexp,nmexp1* AS *var1$* [,*nmexp2* AS *var2$*. . .]

> ***nmexp*** **specifies a random access file buffer, *nmexp*=1,2,...,15.**
>
> ***nmexp1*** **specifies the length of the first field.**
>
> ***var1$*** **defines a variable name for the first field.**
>
> ***nmexp2*** **specifies the length of the second field.**
>
> ***var2$*** **defines a variable name for the second field.**
>
> **. . . Subsequent *nmexp* AS *var$* pairs define other fields in the buffer.**

Note: The sum of all the field-lengths must not exceed the record length, and should equal the record length.

Before FIELDing a buffer, you must use an OPEN statement to assign that buffer to a particular disk file (you must use random access mode). Then use the FIELD statement to organize a random file buffer so that you can pass data from BASIC to disk storage and vice-versa.

Each random file buffer has up to 256 bytes which can store data for transfer from disk storage to BASIC or from BASIC to disk. (When variable-length files are used, maximum may be from 1 to 256.) However, you need a way to access this buffer from BASIC so that you can either read the data it contains or place new data in it. The FIELD statement provides the means of access.

You may use the FIELD statement any number of times to ''re-organize'' a file buffer. FIELDing a buffer does not clear the contents of the buffer; only the means of accessing the buffer (the field names) are changed. Furthermore, two or more field names can reference the same area of the buffer.

Examples

```
FIELD 1, 128 AS A$, 128 AS B$
```

This statement tells BASIC to assign the first 128 bytes of the buffer to the string variable A$ and the remaining 128 bytes to B$. If you now print A$ and B$, you will see the contents of the buffer. Of course, this value would be meaningless unless you have used GET to read a 256-byte record from disk.

Note: All data — both strings and numbers — must be placed into the buffer in string form. There are three pairs of functions (MKI$/CVI,MKS$/CVS,MKD$/CVD) for converting numbers to strings and vice-versa. See ''Functions'' below.

```
FIELD 3, 16 AS NM$, 25 AS AD$, 10 AS CY$, 2 AS ST$,7 AS ZP$
```

The first 16 bytes of buffer 3 are assigned the buffer name NM$; the next 25, AD$; the next 10, CY$; the next 2, ST$ and the next 7, ZP$. The remaining 196 bytes of the buffer are not fielded at all.

More on field names

Field names, like NM$,AD$,CY$,ST$, and ZP$, are not string variables in the ordinary sense. They do not consume the string space available to BASIC.

Instead, they point to the buffer field which you assigned with the FIELD statement. That's why you can use:

```
100 FIELD 1,255 AS A$
```

without worrying about whether 255 bytes of string space are available for A$.

If you use a buffer field name on the left side of an ordinary assignment statement, that name will no longer point to the buffer field; therefore, you won't be able to access that field using the previous field name.

For example,

```
A$=B$
```

nullifies the effect of the FIELD statement above (line 100).

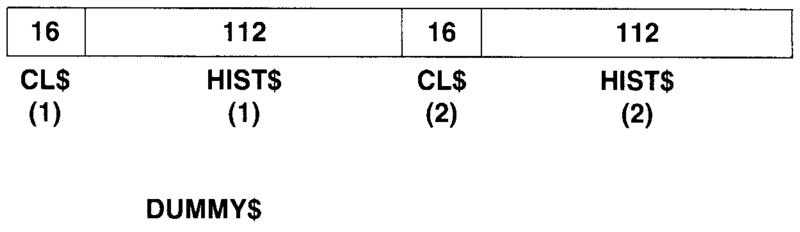
During random input, the GET statement places data into the 255-byte buffer, where it can be accessed using the field names assigned to that buffer. During random output, LSET and RSET place data into the buffer, so you can then PUT the buffer contents into a disk file.

Often you'll want to use a dummy variable in a FIELD statement to ''pass over'' a portion of the buffer and start fielding it somewhere in the middle. For example:

```
FIELD 1, 16 AS CLIENT$(1), 112 AS HIST$(1)
FIELD 1, 128 AS DUMMY$, 16 AS CLIENT$(2), 112 AS HIST$(2)
```

In the second FIELD statement, DUMMY$ serves to move the starting position of CLIENT$(2) to position 129. In this manner, two identical ''subrecords'' are defined on buffer number 1. We won't actually use DUMMY$ to place data into the buffer or retrieve it from the buffer.

The buffer now looks like this:



GET
Read a Record from Disk—Random Access

GET *nmexp1*[,*nmexp2*]

nmexp1 **specifies a random access file buffer,** *nmexp1*=**1,2,...,15.**

nmexp2 **specifies which record to GET in the file; if omitted, the current record will be read.**

This statement gets a data record from a disk file and places it in the specified buffer. Before GETting data from a file, you must open the file and assign a buffer to it. That is, a statement like:

OPEN "R",*nmexp1*,*filespec*

is required *before* the statement:

GET *nmexp1*,*nmexp2*

GET tells BASIC to read record *nmexp2* from the file and place it into the *nmexp1* buffer. If you omit the record number in GET, BASIC will read the current record.

The "current record" is the record whose number is one higher than that of the last record accessed. The first time you access a file via a particular buffer, the current record is set equal to 1.

For example:

Program statement	Effect
1000 OPEN"R",1,"NAME/BAS"	Open NAME/BAS for random access using buffer 1
1010 FIELD 1,...	Structure buffer
1020 GET 1	GET record 1 into buffer 1
1025 REM...ACCESS BUFFER	
1030 GET 1,30	GET record 30 into buffer 1
1035 REM...ACCESS BUFFER	
1040 GET 1,25	GET record 25 into buffer 1
1046 REM...ACCESS BUFFER	
1050 GET1	GET record 26 into buffer 1

If you are using variable-length records (not fixed-length), an attempt to GET past the end of file will produce an error.

If you are using fixed-length records, the same attempt will return a null record and no error will occur. To prevent this from occurring, you can use the LOF function to determine the number of the highest numbered record.

PUT
Write a Record to Disk—Random Access

PUT *nmexp1*[,*nmexp2*]

nmexp1 specifies a random access file buffer, _nmexp_=1,2,...,15.

nmexp2 specifies the record number in the file, _nmexp2_ is the record you want to write. If _nmexp2_ is omitted, the current record number is assumed.

This statement moves data from a file's buffer into a specified place in the file. Before PUTting data in a file, you must:

1. OPEN the file, thereby assigning a buffer and defining the access mode (must be R);

2. FIELD the buffer, so you can

3. place data into the buffer with LSET and RSET statements.

When BASIC encounters the statement:

PUT *nmexp,nmexp2*

it does the following:

- Gets the information needed to access the disk file
- Checks the access mode for this buffer (must be R)
- Acquires more disk space for the file if necessary to accommodate the record indicated by *nmexp2*
- Copies the buffer contents into the specified record of the disk file
- Updates the current record number to equal *nmexp2* + *1*

The "current record" is the record whose number is one higher than the last record accessed. The first time you access a file via a particular buffer, the current record is set equal to 1.

If the record number you PUT is higher than the end-of-file record number, then *nmexp2* becomes the new end-of-file record number.

LSET and RSET Place Data in a Random Buffer Field

LSET *var$* = *exp$* and RSET *var$*= *exp$*

***var$* is a field name.**

***exp$* contains the data to be placed in the buffer field named by *var$*.**

These two statements let you place character-string data into fields previously set up by a FIELD statement.

For example, suppose NM$ and AD$ have been defined as field names for a random file buffer. NM$ has a length of 18 characters, and AD$ has a length of 25 characters.

Now we want to place the following information into the buffer fields so it can be written to disk:

```
name:      JIM CRICKET, JR.
address:   2000 EAST PECAN ST.
```

This is accomplished with the two statements:

```
LSET NM$="JIM CRICKET,JR. "
LSET AD$="2000 EAST PECAN ST. "
```

This puts the data in the buffer as follows:

```
| JIM CRICKET,JR.       |     | 2000 EAST PECAN ST.         |
          NM$                             AD$
```

Note that filler spaces were placed to the right of the data strings in both cases. If we had used RSET instead of LSET statements, the filler spaces would have been placed on the left. This is the **only** difference between LSET and RSET.

For example:

```
RSET NM$="JIM CRICKET,JR. "
RSET AD$="2000 EAST PECAN ST. "
```

places data in the fields as follows:

```
|       JIM CRICKET,JR. |     |         2000 EAST PECAN ST. |
          NM$                             AD$
```

If a string item is too large to fit in the specified buffer field, it is always truncated on the right. That is, the extra characters on the right are ignored.

CVD, CVI and CVS
Restore String to Numeric Form

CVD(*exp$*)

exp$_ defines an eight-character string; _exp$ is typically the name of a buffer field containing a numeric string. If LEN(_exp$_)<8, an ILLEGAL FUNCTION CALL error occurs; if LEN(_exp$_)>8, only the first eight characters are used.

CVI(*exp$*)

exp$_ defines a two-character string; _exp$ is typically the name of a buffer field containing a numeric string. If LEN(_exp$_)<2, an ILLEGAL FUNCTION CALL error occurs; if LEN(_exp$_)>2, only the first two characters are used.

CVS(*exp$*)

***exp$*** **defines a four-character string; *exp$* is typically the name of a buffer field containing a numeric string. If LEN(*exp$*)<4, an ILLEGAL FUNCTION CALL error occurs; if LEN(*exp$*)>4, only the first four characters are used.**

These functions let you restore data to numeric form after it is read from disk. Typically the data has been read by a GET statement, and is stored in a random-access file buffer.

The functions CVD, CVI, and CVS are inverses of MKD$, MKI$, and MKS$, respectively.

For example, suppose the name GROSSPAY$ references an eight-byte field in a random-access file buffer, and after GETting a record, GROSSPAY$ contains a MKD$ representation of the number 13123.38.

Then the statement:

```
PRINT CVD(GROSSPAY$)-TAXES
```

prints the result of the difference, 13123.38 – TAXES. Whereas the statement:

```
PRINT GROSSPAY$-TAXES
```

will produce a TYPE MISMATCH error, since string values cannot be used in arithmetic expressions.

Using the same example, the statement

```
A#=CVD(GROSSPAY$)
```

assigns the numeric value 13123.38 to the double-precision variable A#.

EOF
End-Of-File Detector

EOF(*nmexp*)

nmexp **specifies a file buffer, *nmexp*=1,2,...,15.**

This function checks to see whether all characters up to the end-of-file marker have been accessed, so you can avoid INPUT PAST END errors during sequential input.

Assuming *nmexp* specifies an open file, then EOF*(nmexp)* returns 0 (false) when the EOF record has not yet been read, and −1 (true) when it has been read.

Examples

```
IF EOF(5) THEN PRINT"END OF FILE"FILENM$
IF EOF(NM%) THEN CLOSE NM%
```

The following sequence of lines reads numeric data from DATA/TXT into the array A( ). When the last data character in the file is read, the EOF test in line 30 "passes," so the program branches out of the disk access loop, preventing an INPUT PAST END error from occurring. Also note that the variable I contains the number of elements input into array A( ).

```
5 DIM A(100) 'ASSUMING THIS IS A SAFE VALUE
10 OPEN "I",1, "DATA/TXT"
20 I%=0
30 IF EOF(1) THEN 70
40 INPUT#1,A(I%)
50 I%=I%+1
60 GOTO 30
70 REM PROGRAM CONTINUES HERE AFTER DISK INPUT
```

LOC
Get Current Record Number

LOC***(file number)***

file number **is a numeric expression specifying the buffer for a currently-open file.**

LOC is used to determine the current record number, i.e., the number of the last record read since the file was opened. LOC is only valid after a GET.

Example

```
PRINT LOC(1)
```

Sample Program

```
1310 A$ = "WILLIAM WILSON"
1320 GET 1, X: X=X+1
1330 IF N$ = A$ THEN PRINT "FOUND IN RECORD" LOC(1): CLOSE:
     END
1340 GOTO 1320
```

This is a portion of a program. Elsewhere the file has been opened and fielded. N$ is a field variable. If N$ matches A$ the record number in which it was found is printed.

LOF
Get End-Of-File Record Number

LOF*(nmexp)*

nmexp **specifies a random access buffer** *nmexp*=**1,2,...,15.**

This function tells you the number of the last, i.e., highest numbered, record in a file. It is useful for both sequential and random access.

For example, during random access to a pre-existing file, you often need a way to know when you've read the last valid record. LOF provides a way.

LOF is valid as soon as a previously created file is opened. If a file is extended, LOF is not valid until a GET is executed.

Examples:

```
10 OPEN "R",1,"UNKNOWN/TXT"
20 FIELD 1,255 AS A$
30 FORI%=1 TO LOF(1)
40 GET 1,I%
50 PRINT A$
60 NEXT
```

In line 30, LOF(1) specifies the highest record number to be accessed.

Note: If you attempt to GET record numbers beyond the end-of-file record, BASIC simply fills the buffer with hexadecimal zeros, and no error is generated.

When you want to add to the end of a file, LOF tells you where to start adding:

```
100 I%=LOF(1)+1 'HIGHEST EXISTING RECORD
110 PUT 1,I%    'ADD NEXT RECORD
```

MKD$, MKI$, and MKS$
Convert Data, Numeric-to-String

MKD$(*nmexp*)

**   *nmexp* is evaluated as a double-precision number.**

MKI$(*nmexp*)

**   *nmexp* is evaluated as an integer, −32768< = *nmexp*<32768; if *nmexp* exceeds this range, an ILLEGAL FUNCTION CALL error occurs. Any fractional component in *nmexp* is truncated.**

MKS$(*nmexp*)

**   *nmexp* is evaluated as a single-precision number.**

These functions change a number to a "string." Actually the byte values which make up the number are not changed; only one byte, the internal data-type specifier, is changed, so that numeric data can be placed in a string variable.

That is:

MKD$ returns an eight-byte string.
MKI$ returns a two-byte string.
MKS$ returns a four-byte string.

Examples

```
LSET TALLY$=MKI$(I%)
```

Field name TALLY$ would now contain a two-byte representation of the integer I%.

```
A$=MKI$(8/I)
```

A$ becomes a two-byte representation of the integer portion of 8/I. Any fractional portion is ignored. Note that A$ in this case is a normal string variable, not a buffer-field name.

Suppose BASEBALL/BAT (a non-standard file extension) has been opened for random access using buffer 2, and the buffer has been FIELDed as follows:

field:	NM$	YRS$	AVG$	HR$	AB$	ERNING$
length:	16	2	4	2	4	4

NM$ is intended to hold a character string; AVG$, AB$ and ERNING$, converted single-precision values; YRS$ and HR$, converted integers.

Suppose we want to write the following data record:

SLOW LEARNER played 38 years; lifetime batting average .123; career homeruns, 11; at bats, 32768;...,earnings − 13.75.

Then we'd use the make-string functions as follows:

```
1000 LSET NM$="SLOW LEARNER"
1010 LSET YRS$=MKI$(38)
1020 LSET AVG$=MKS$(.123)
1030 LSET HR$=MKI$(11)
1040 LSET AB$=MKS$(32768)
1050 LSET ERNING$=MKS$(-13.75)
```

After this sequence, you can write SLOW LEARNER's information to disk with the PUT statement. When you read it back from disk with GET, you will need to restore the numeric data from string to numeric form, using CVI and CVS functions.

Methods of Access

Disk BASIC provides two means of file access:

- Sequential — in which you start reading or writing data at the beginning of a file; subsequent reads or writes are done at following positions in the file.
- Random — in which you start reading or writing at any record you specify. (Random access is also called direct access.)

Sequential access is stream-oriented; that is, the number of characters read or written can vary, and is usually determined by delimiters in the data. Random access is record-oriented; that is, data is always read or written in fixed-length blocks called records.

To do any input/output to a disk file, you must first open the file. When you open the file, you specify what kind of access you want:

- O for sequential output
- I for sequential input
- R for random input/output
- E (Extend) for sequential output starting at the end of file.

You also assign a file buffer for BASIC to use during file accesses. This number can be from 1 to 15, but must not exceed the number of concurrent files you requested when you started BASIC from TRSDOS. For example, if you started BASIC with 3 files, you can use buffer numbers 1, 2, and 3. Once you assign a buffer number to a file, you cannot assign that number to another file until you Close the first file.

Examples

```
OPEN "O", 1, "TEST"
```

Creates a sequential output file named TEST on the first available drive; if TEST already exists, its previous contents are lost. Buffer 1 will be used for this file.

```
OPEN "I", 2, "TEST"
```

Opens TEST for sequential input, using buffer 2.

```
OPEN "R", 1, "TEST"
```

Opens TEST for direct access, using buffer 1. If TEST does not exist, it will be created on the first available drive. Since record length is not specified, 256-byte records will be used.

```
OPEN "R", 1, "TEST", 40
```

Same as preceding example, but 40-byte records will be used.

```
OPEN "E", 1, "TEST"
```

Opens TEST sequentially for write and positions to EOF.

Sequential Access

This is the simplest way to store data in and retrieve it from a file. It is ideal for storing free-form data without wasting space between data items. You read the items back in the same order in which they were written.

There are several important points to keep in mind.

1. You must start writing at the beginning of the file. If the data you are seeking is somewhere inside, you have to read your way up to it.
2. Each time you Open a file for sequential output, the file's previous contents are lost, unless you use "E" instead of "O" for the mode.
3. To update (change) a sequential file, read in the file and write out the updated data to a *new* output file.
4. Data written sequentially usually includes delimiters (markers) to signify where each data item begins and ends. To read a file sequentially, you must know ahead of time the format of the data. For example: Does the file consist of lines of text terminated with carriage returns? Does it consist of numbers separated by blank spaces? Does it consist of alternating text and numeric information?
5. Sequential files are always written as ASCII-coded text, one byte for each character of data. For example, the number:

   ```
    1.2345
   ```

   requires 8 bytes of disk storage, including the leading and trailing blanks that are supplied. The text string:

   ```
   JOHNSON, ROBERT
   ```

   requires 15 bytes of disk storage.
6. Sequential files are always written with a record length of 256.

Sequential Output: An Example

Suppose we want to store a table of English-to-metric conversion constants:

English unit	Metric equivalent
1 inch	2.54001 centimeters
1 mile	1.60935 kilometers
1 acre	4046.86 sq. meters
1 cubic inch	0.01638716 liter
1 U.S. gallon	3.785 liters
1 liquid quart	0.9463 liter
1 lb (avoir)	0.45359 kilogram

First we decide what the data image is going to be. Let's say we want it to look like this:

english unit , metric unit, factor (ENTER)

For example, the stored data would start out:

```
IN->CM, 2.54001  (ENTER)
```

The following program will create such a data file.

Note: X'0D' represents a carriage return.

```
10 OPEN "O",1,"METRIC/TXT"
20 FOR I%=1 TO 7
30     READ UNIT$, FACTR
40     PRINT#1, UNIT$; ","; FACTR
50 NEXT
60 CLOSE
70 DATA IN->CM, 2.54001, MI->KM, 1.60935, ACRE->SQ.KM,
   4046.86 E-6
80 DATA CU.IN->LTR, 1.638716E-2, GAL->LTR, 3.785
90 DATA LIQ.QT->LTR, 0.9463, LB->KG, 0.45359
```

Line 10 creates a disk file named METRIC/TXT, and assigns buffer 1 for sequential output to that file. The extension /TXT is used because sequential output always stores the data as ASCII-coded text.

Note: If METRIC/TXT already exists, line 10 will cause all its data to be lost. Here's why: Whenever a file is opened for sequential output, the end-of-file (EOF) is set to the beginning of the file. In effect, TRSDOS "forgets" that anything has ever been written beyond this point. To avoid this, you could use E instead of O in line 10.

Line 40 prints the current contents of UNIT$ and FACTR to the file. Since the spring items do not contain delimiters, it is not necessary to print explicit quotes around them. The explicit comma is sufficient.

Line 60 closes the file. The EOF is at the end of the last data item, i.e., 0.45359, so that later, during input, BASIC will know when it has read all the data.

Sequential Input: An Example

The following program reads the data from METRIC/TXT into two "parallel" arrays, then asks you to enter a conversion problem.

```
5 CLEAR 500
10 DIM UNIT$(9), FACTR(9)    'allows for up to 10 data pairs
20 OPEN"I",1,"METRIC/TXT"
25 I%=0
30 IF EOF(1) THEN 70
40 INPUT#1, UNIT$(I%),FACTR(I%)
```

```
50 I%=I%+1
60 GOTO 30
70 CLOSE    '   Conversion factors have been read-in
100 CLS: PRINT TAB(5)"*** English to Metric Conversions ***"
110 FOR ITEM%=0 TO I%-1
120     PRINT TAB(9);USING"(## )   %          %    ";ITEM%,
    UNIT$(ITEM%)
130 NEXT
140 PRINT @ 704, "Which conversion (0-6)";
150 INPUT CHOICE%
160 INPUT"Enter English quantity";V
170 PRINT"The Metric equivalent is" V*FACTR(CHOICE%)
180 INPUT"Press <ENTER> to continue";X
190 PRINT @ 704, CHR$(31)    'clear to end of frame
200 GOTO 140
```

Line 20 opens the file for sequential input. Input begins at the beginning of the file.

Line 30 checks to see that the end-of-file record hasn't been reached. If it has, control branches from the disk input loop to the part of the program that uses the newly acquired data.

Line 40 reads a value into the string array UNIT$( ), and a number into the single-precision array FACTR( ). Note that this INPUT list parallels the PRINT# list that created the data file (see the section "Sequential Output: An Example"). This parallelism is not required, however. We could just as successfully have used:

```
40 INPUT#1, UNIT$(I%): INPUT#1,FACTR(I%)
```

How to update a file

Suppose you want to add more entries into the English-Metric conversion file. You could simply re-Open the file with mode = E and PRINT# the extra data. Or, you might want to leave the old file intact and output a new file:

1. Open the file for sequential input (Mode = I)
2. Open another new data file for sequential output (Mode = O)
3. Input a block of data and update the data as necessary
4. Output the data to the new file
5. Repeat steps 3 and 4 until all data has been read, updated, and output to the new file; then go to step 6
6. Close both files

Sequential Line Input: An Example

Using the line-oriented input, you can write programs that edit other BASIC program files: renumber them, change LPRINTS to PRINTS, etc.—as long as these "target" programs are stored in ASCII format.

The following program counts the number of lines in any ASCII—format BASIC disk file with the extension /TXT.

```
10 CLEAR 300
20 INPUT"WHAT IS THE NAME OF THE PROGRAM"; PROG$
30 IF INSTR(PROG$,"/TXT")=0 THEN 110 'require /TXT extension
40 OPEN"I", 1, PROG$
50 I%=0
60 IF EOF(1) THEN 90
70 I%=I%+1: LINE INPUT#1, TEMP$
80 GOTO 60
90 PRINT PROG$" IS" I% "LINES LONG."
100 CLOSE: GOTO 20
110 PRINT "FILESPEC MUST INCLUDE THE EXTENSION '/TXT'"
120 GOTO 20
```

For BASIC programs stored in ASCII, each program line ends with a carriage return character not preceded by a line feed. So the LINE INPUT in line 70 automatically reads one entire line at a time, into the variable TEMP$. Variable I% actually does the counting.

To try out the program, first save any BASIC program using the A (ASCII) option (See SAVE). Use the extension /TXT.

Random Access Techniques

Random access offers several advantages over sequential access:

- Instead of having to start reading at the beginning of a file, you can read any record you specify.
- To update a file, you don't have to read in the entire file, update the data, and write it out again. You can rewrite or add to any record you choose, without having to go through any of the other records.
- Random access is more efficient — data takes up less space and is read and written faster.
- Opening a file for direct access allows you to write and read from the file via the same buffer.
- Random access provides many powerful statements and functions to structure your data. Once you have set up the structure, direct input/output becomes quite simple.

The last advantage listed above is also the "hard part" of direct access. It takes a little extra thought.

For the purposes of direct access, you can think of a disk file as a set of boxes — like a wall of post-office boxes. Just like the post office receptacles, the file boxes are numbered. We call these boxes "records."

You can place data in any record, or read the contents of any record, with statements like:

`PUT 1,5` write buffer-1 contents to record 5

`GET 1,5` read the contents of record 5 into buffer-1

In **Figure 14**, we assume a record length of 256.

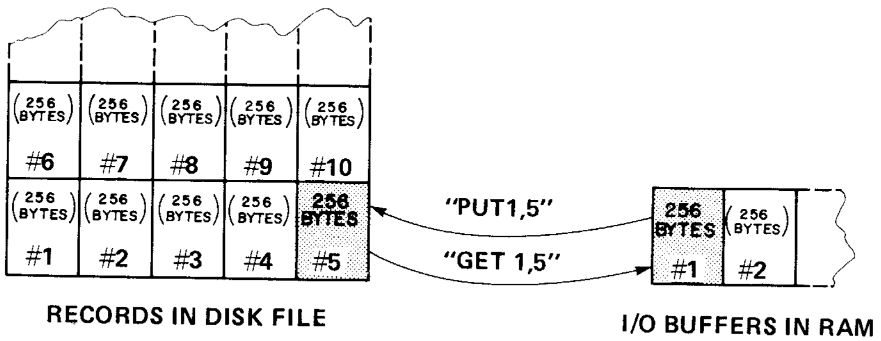
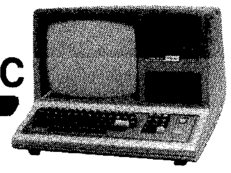


Figure 14. GET and PUT.



The buffer is a waiting area for the data. Before writing data to a file, you must place it in the buffer assigned to the file. After reading data from a file, you must retrieve it from the buffer.

As you can see from the sample PUT and GET statements above, data is passed to and from the disk in records. The size of each record is determined by an Open statement.

Storing Data in a Buffer

You must place the entire record into the buffer before putting its contents into the disk file.

This is accomplished by 1) dividing the buffer up into fields and naming them, then 2) placing the string or numeric data into the fields.

For example, suppose we want to store a glossary on disk. Each record will consist of a word followed by its definition. We start with:

```
100 OPEN"R", 1, "GLOSSARY/BAS"
110 FIELD 1, 16 AS WD$, 240 AS MEANING$
```

Line 100 opens a file named GLOSSARY/BAS (creates it if it doesn't already exist); and gives buffer 1 direct access to the file.

Line 110 defines two fields onto buffer 1:

WD$	consists of the first 16 bytes of the buffer;
MEANING$	consists of the last 240 bytes.

WD$ and MEANING$ are now **field-names**

What makes field names different? Most string variables point to an area in memory called the string space. This is where the value of the string is stored.

Field names, on the other hand, point to the buffer area assigned in the FIELD statement. So, for example, the statement:

```
10 PRINT WD$; ":"; MEANING$
```

displays the contents of the two buffer fields defined above.

These values are meaningless unless we first place data in the buffer. LSET, RSET and GET can all be used to accomplish this function. We'll start with LSET and RSET, which are used in preparation for disk output.

Our first entry is the word "left-justify" followed by its definition.

```
100 OPEN"R", 1, "GLOSSARY/BAS"
110 FIELD 1, 16 AS WD$, 240 AS MEANING$
120 LSET WD$="LEFT-JUSTIFY"
130 LSET MEANING$="TO PLACE A VALUE IN A FIELD FROM LEFT TO
    RIGHT; IF THE DATA DOESN'T FILL THE FIELD, BLANKS ARE
    ADDED ON THE RIGHT; IF THE DATA IS TOO LONG, THE EXTRA
    CHARACTERS ON THE RIGHT ARE IGNORED, LSET IS A LEFT-
    JUSTIFY FUNCTION."
```

Line 120 left-justifies the value in quotes into the first field in buffer 1. Line 130 does the same thing to its quoted string.

Note: RSET would place filler-blanks to the *left* of the item. Truncation would still be on the right.

Now that the data is in the buffer, we can write it to disk with a simple PUT statement:

```
140 PUT 1,1
150 CLOSE
```

This writes the first record into the file GLOSSARY/BAS.

To read and print the first record in GLOSSARY/BAS, use the following sequence:

```
160 OPEN"R", 1, "GLOSSARY/BAS"
170 FIELD 1, 16 AS WD$, 240 AS MEANING$
180 GET 1,1
190 PRINT WD$: PRINT MEANING$
200 CLOSE
```

Line 160 and 170 are required only because we closed the file in line 150. If we hadn't closed it, we could go directly to line 180.

Random Access: A General Procedure

The previous example shows the necessary sequences to read and write using random access. But it does not demonstrate the primary advantages of this form of access — in particular, it doesn't show how to update existing files by going directly to the desired record.

The program below, GLOSSACC/BAS, develops the glossary example to show some of the techniques of random access for file maintenance. But before looking at the program, study this general procedure for creating and maintaining files via random access.

Step	See GLOSSACC/BAS, Line Number
1. Open the file	110
2. Field the buffer	120
3. Get the record to be updated	140
4. Display current contents of the record (use CVD, CVI, CVS before displaying numeric data)	145-170
5. LSET and RSET new values into the fields (use MKD$, MKI$, MKS$ with numeric data before setting it into the buffer)	210-230
6. PUT the updated record	240
7. To update another record, continue at step 3. Otherwise, go to step 8.	250-260
8. Close the file	270

```
10 REM ...... GLOSSACC/BAS ...
100 CLS : CLEAR 300
110 OPEN "R", 1, "GLOSSARY/BAS"
120 FIELD 1, 16 AS WD$, 238 AS MEANING$, 2 AS NX$
130 INPUT "WHAT RECORD DO YOU WANT TO ACCESS"; R%
140 GET 1, R%
145 NX% = CVI(NX$)    'SAVE LINK TO NEXT ALPHABETICAL ENTRY
150 PRINT "WORD :     "WD$
160 PRINT "DEF'N : " : PRINT MEANING$
170 PRINT "NEXT ALPHABETICAL ENTRY: RECORD #:" NX% : PRINT
180 W$ = "" : INPUT "TYPE NEW WORD <ENTER> OR <ENTER> IF OK";
    W$
190 D$ = "" : PRINT "TYPE NEW DEF'N <ENTER> OR <ENTER> IF
    OK?" : LINE INPUT D$
200 INPUT "TYPE NEW SEQUENCE NUMBER OR <ENTER> IF OK"; NX%
210 IF W$ <> "" THEN LSET WD$ = W$
220 IF D$ <> "" THEN LSET MEANING$ = D$
```

```
230 LSET NX$ = MKI$ (NX%)
240 PUT 1, R%
245 R% = NX% 'USE NEXT ALPHA. LINK AS DEFAULT FOR NEXT RECORD
250 CLS : PRINT " TYPE <ENTER> TO READ NEXT ALPHA. ENTRY,":
    PRINT" OR RECORD # <ENTER> FOR SPECIFIC ENTRY,": INPUT "
    OR 0 <ENTER> TO QUIT"; R%
260 IF 0<R% THEN 140
270 CLOSE
280 END
```

Notice we've added a field, NX$, to the record (line 120). NX$ will contain the number of the record which comes next in alphabetical sequence. This enables us to proceed alphabetically through the glossary, provided we know which record contains the entry which should come first.

For example, suppose the glossary contains:

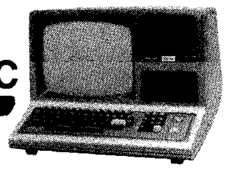
record#	word (WD$)	defn,	pointer to next alpha. entry (NX$)
1	LEFT-JUSTIFY	...	3
2	BYTE	...	4
3	RIGHT-JUSTIFY	...	0
4	HEXADECIMAL	...	1

When we read record 2 (BYTE), it tells us that record 4 (HEXADECIMAL) is next, which then tells us record 1 (LEFT-JUSTIFY) is next, etc. The last entry, record 3 (RIGHT-JUSTIFY), points us to zero, which we take to mean ''The End.''

Since NX$ will contain an integer, we have to first convert that number to a two-byte string representation, using MKI$ (line 230 above).

The following program displays the glossary in alphabetical sequence:

```
300 REM ... GLOSSOUT/BAS ...
310 CLS : CLEAR 300
320 OPEN "R", 1, "GLOSSARY/BAS"
330 FIELD 1, 16 AS WD$, 238 AS MEANING$, 2 AS NX$
340 INPUT "WHICH RECORD IS FIRST ALPHABETICALLY"; N%
350 GET 1, N%
360 PRINT : PRINT WD$
370 PRINT MEANING$
380 N% = CVI(NX$)
390 INPUT "PRESS <ENTER> TO CONTINUE"; X
400 IF N% <> 0 THEN 350
410 CLOSE
420 END
```



Disk BASIC Error Codes/Messages

51	Field overflow
52	Internal error
53	Bad file number
54	File not found
55	Bad file mode
58	Disk I/O error
62	Disk full
63	Input past end
64	Bad record number
65	Bad file name
67	Direct statement in file
68	Too many files
69	Disk write-protect
70	File access

Note: Disk errors cannot be simulated via the ERROR statement.